TRS-XENIX System Software Development

TRS-XENIX SYSTEM

SOFTWARE DEVELOPMENT

ACKNOWLEDGEMENTS

This manual builds on the writing of many others. In many cases, the content here is identical, in whole or in part, to papers and manuals written at Bell Laboratories. In particular, Chapter 2 and Appendix B are adapted from papers written by Brian Kernighan and D.M. Ritchie. Chapters 5 and 10 are adapted from papers written by S.C. Johnson. Chapter 6 is derived from a paper by S.I. Feldman, Chapter 7 from a a paper by J.F. Marazano and S.R. Bourne, and Chapter 9 from a paper by M.E. Lesk and E. Schmidt. In addition, Appendix A is adapted from material written by Bill Joy and Mark Horton, while at the University of California at Berkeley. The work of those mentioned above, and countless others, is gratefully acknowledged.

CONTENTS

9.0 Lex: A Lexical Analyzer

10.0 YACC: A Compiler-Compiler

CHAPTER 1

INTRODUCTION

CONTENTS

1.1 Overview

One of the primary uses of the XENIX system is as an environment for software development. This manual describes this programming environment and the available tools. Since nearly all of the XENIX system is written in the C programming language, C is the ideal language for creating new XENIX applications. However, no attempt is made here to teach C programming. For that, see the excellent tutorial and reference The C Programming Language, by Kernighan and Ritchie. For more information about the basic concepts and software that underly XENIX itself, see the XENIX Fundamentals manual.

1.2 Manual Organization

This manual is organized as follows:

CHAPTER 1: Introduction

The chapter you are now reading contains a word about the development of software on the XENIX system

CHAPTER 2: Xenix Programming

Discusses the standard XENIX environment and how this environment can be accessed either from C or from assembly language.

CHAPTER 3: Software Tools

Describes each of the tools that you are likely to use either directly or indirectly, in programming on the XENIX system, with emphasis on how the the software tools discussed in this manual fit together.

CHAPTER 4: Cc: The C Compiler

Describes use of the XENIX C compiler, cc. Also describes the preprocessing, linking, and assembly stages in compiling C programs to executable files.

CHAPTER 5: Lint: The C Program Checker

Describes use of lint, the XENIX C program checker. Lint analyzes C program syntax and language usage, reporting anomalies to the user.

CHAPTER 6: Make: A Program Maintainer

Describes use of **make**, a program for controlling software generation, update, and installation.

CHAPTER 7: ADB: A Program Debugger

Describes use of the debugger, **ADB**, a program for debugging and analyzing both programs while they execute.

CHAPTER 8: As: The XENIX Assembler

Describes how **as**, the XENIX assembler can be used to assemble machine language programs and routines.

CHAPTER 9: Lex: A Lexical Analyzer

Describes use of **lex**, a lexical analyzer useful in reading input languages.

CHAPTER 10: YACC: A Compiler-Compiler

Describes use of YACC, a complex utility for creating language translators. Useful in conjunction with **lex**, above.

APPENDIX A: The C Shell

Describes use of the alternate shell command interpreter, **csh**. The C shell command language has a syntax similar to that of the C programming language. Aliases and a command history mechanism are also provided.

APPENDIX B: M4

Describes use of the macro preprocessor, **M4**.

APPENDIX C: C Program Portability

Explains how to write C programs that are portable across different processors and XENIX systems.

1.3 Notational Conventions

Throughout this manual, the following notational conventions are used:

boldface — Command names are given in boldface in the text of this manual; no boldface occurs in displays, except in syntax specifications for literal text. For example, **ls**, **date**, and **cd** are all the names of commands that you might type at the keyboard, and therefore all are in bold. An exception to this rule occurs for long chapters about a single command. In this case, the command name is made less conspicuous by either underlining or capitalization.

underlining — All filenames and pathnames are underlined. For example, text.file is a filename and /usr/mary is a pathname. Most command arguments are underlined as well, although in some cases these are in boldface. Words and phrases also may be underlined for emphasis. References to entries in the XENIX Reference Manual are underlined and include a section number in parentheses. For example, ls(1) refers to the entry for the ls command in Section 1, "Commands".

[brackets] — Brackets enclose optional arguments in syntax specifications.

<angle-brackets> — Angle brackets enclose the names of control characters and special function keys. Examples are <CONTROL-D>, <CONTROL-S>, <RETURN>, <INTERRUPT>, and <BKSP>.

ellipses... — Ellipses are used to indicate one or more entries of an argument in a syntax specification. For example, in the following syntax for the **mail** command, the ellipses indicate that one or more persons can be sent mail:

```
mail person ...
```

quotation marks — Quotation marks are used to set off multiple keystroke input. For example,

"ls -la ; date" is an example of a command line appearing in the body of the text.

Common abbreviations for ASCII characters are listed below:

<ESC>	Escape, Control-[
<RETURN>	Carriage return, Control-M
<LF>	Newline, Linefeed, Control-J
<NL>	Newline, Linefeed, Control-J
<BKSP>	Backspace, Control-H
<TAB>	Tab, Control-I
<BELL>	Bell, Control-G
<FF>	Formfeed, Control-L
<SPACE>	Space, octal 040
<DEL>	Delete, octal 0177

CHAPTER 2

XENIX PROGRAMMING

CONTENTS

2.1 Introduction

The C programming language is designed to be used in a computing environment. Because of the power and flexibility of the XENIX environment, it is important for the programmer to to take advantage of its many capabilities. For example, from within some C programs, you may want to execute other programs, or make calls to perform system functions. Or, you may want to write assembly language routines that interface to C programs. Before you can perform any of these programming tasks, you must know the environment. In the case of the XENIX system, this environment includes low-level system calls, available C libraries, and compiler calling conventions. Because you may also want to write C programs that are portable to other XENIX systems and other processors, a section in this chapter discusses portable C programming.

2.2 The C Interface To The XENIX System

This section shows how to interface C programs to the XENIX system, either directly or through the standard I/O library. The topics discussed include:

- Handling command arguments
- Rudimentary I/O
- The standard input and output
- The standard I/O library
- File system access
- Low-level I/O: open, read, write, close, seek
- Processes: exec, fork, pipes
- Signals and interrupts

2.2.1 Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the 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.)

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

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

argv is a pointer to an array whose individual 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 them all.

The argument count and the arguments are parameters to main. If you want to keep them so other routines can get at them, you must copy them to external variables.

2.2.2 The Standard

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function getchar returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention:

```
prog <file
```

This causes prog to read file instead of the terminal. The program itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the "pipe" mechanism. For example

```
otherprog | prog
```

provides the standard input for prog from the standard output 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.

Similarly, putchar(c) puts the character c on the "standard output," which is also by default the terminal. The output can be captured on a file by using >. If prog uses putchar,

```
prog >outfile
```

writes the standard output on outfile instead of the terminal. Outfile is created if it doesn't exist; if it already exists, its previous contents are overwritten.

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; it 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 XENIX pipe facility is used to connect the output of one program to the input of the next. 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. It causes the C compiler to read 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:

```
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. Status returns are discussed later in more detail.

2.2.3 The Standard I/O Library

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.

In this section, we will discuss the basics of the standard I/O library. Section 2.3 contains a more complete description of its capabilities.

2.2.3.1 File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. 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

```
wc x.c y.c
```

prints 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 file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be opened by the standard library function fopen. 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, which is a type name, like int, not a structure tag.

The actual call to fopen in a program is

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

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 only allowable modes are read (r), write (w), or append (a).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. 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 when you don't have permission). If there is any error, fopen returns the null pointer value NULL (which is defined as zero in stdio.h).

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. For example

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

puts the character c on the file fp and returns c. Getc and putc return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files 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. Stdin, stdout and stderr are pre-defined 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 stand-alone or as part of a larger process.

```
#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. Since there is a limit on the number of files

that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call fclose on an output file-it flushes the buffer in which putc is collecting output. fclose(is called automatically for each open file when a program terminates normally.)

2.2.3.2 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. 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 program actually signals errors in another way, using the function exit to terminate program execution. The argument of exit is available to whatever process called it, so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal 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 causes immediate termination without any buffer flushing; it may be called directly if desired.

2.2.3.3 Miscellaneous I/O Functions

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

Normally output with putc, etc., is buffered (except to stderr); to force it out immediately, use fflush(fp).

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.

fgets(buf, size, fp) copies the next line from 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, etc., will encounter c. Only one character of push-back per file is permitted.

2.2.4 Low-Level I/O

This section describes the bottom level of I/O on the XENIX system. The lowest level of I/O in XENIX provides neither buffering nor any other services; 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.

2.2.4.1 File Descriptors

In the XENIX 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 and 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 concept 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 the standard input, the standard output, and the 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 worrying about opening the files.

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

```
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.

2.2.4.2 Read and Write

All input and output is done by two functions called read and write. For both, the first argument 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 512, which corresponds to a physical block size 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 copies anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512

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, the last read will return a smaller number of bytes to be written by write; 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   0377    /* 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 accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is machine dependent and thus varies from machine to machine.)

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

```
#define CMASK   0377    /* for making char's > 0 */
#define BUFSIZE 512

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);
}
```

2.2.4.3 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 [sic].

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 any 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 re-write old ones.

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

returns a file descriptor if it was able to 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 brand new, creat creates it with the protection mode specified by the pmode argument. In the

XENIX 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 XENIX 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 512
#define PMODE 0644 /* RW for owner, R for group, others */

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);
}

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

There is a limit (typically 20) on the number of files which a program may have open simultaneously. Therefore, 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. Termination of a program via exit or return from the main program closes all open files.

The following function removes the file filename from the file system:

```
unlink (filename)
```

2.2.4.4 Random Access-Seek and 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));
}
```

2.2.4.5 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 cell errno. The meanings of the various error numbers are listed in the introduction to Section II of the XENIX Reference 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 print out the reason for failure. The routine perror will print 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 printed by your program.

2.2.5 Processes

It is often easier to use a program written by someone else than to invent your own. This section describes how to execute a program from within another.

2.2.5.1 The System"

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 time-stamp the output of a program,

```
main()
{
        system("date");
        /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of sprintf may be useful.

Remember than getc and putc normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use fflush; for input, see setbuf in the appendix.

2.2.5.2 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.

The most basic operation is to execute another program without returning, by using the routine execl. To print the date as the last action of a running program, use

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

The first argument to execl is the filename of the command; you have to know where it is found in the file system. 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 place-holder. If the command takes arguments, they are strung out after this; 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 realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply 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, for example if the file can't be found or is not executable. If you don't know where date is located, say

```
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, 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.

2.2.5.3 Control of Processes - 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();
```

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 non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

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

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 non-zero 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, if you want to check it against the value returned by fork.) Finally, this fragment doesn't deal with any funny 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 non-zero 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 pointing at the right files, 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 the exec calls affects 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.

2.2.5.4 Pipes

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

```
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 will 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 will wait until data arrives; if a process writes into a pipe which is too full, it will wait 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 the pipe with a pipe system call; it then forks 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);
                /* disaster has occurred if we get here*/
                _exit(1);
        }
        if (popen_pid == -1)
                return(NULL);
        close(tst(p[READ], p[WRITE]));
        return(tst(p[WRITE], p[READ]));
}
```

The sequence of closes 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. (Yes, this is a bit tricky, but it's a standard idiom.) 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 from 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 is that only a finite number of unwaited-for children can exist for a given parent process, 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.

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 below. As currently written, it shares the same limitation.

2.2.6 Signals and Interrupts

This section is concerned with how to deal gracefully with program faults and with signals and interrupts from the outside world. Since there's nothing very useful that can be done from within C 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**, which is sent when the character is typed; **quit**, generated by the character; **hangup**, caused by hanging up the phone; and **terminate**, generated by the **kill** command. When one of these events occurs, the signal is sent to all processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the **quit** case, a core image file is written for debugging purposes.

The routine which alters the default action is called signal. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address 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);
```

causes interrupts to be 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 has to 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 to allow the program to 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 interrupt are sent to all processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), 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 printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>

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

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

        /* main processing loop */
}

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

The include file setjmp.h declares the type jmp buf an object in which the state can be saved. Sjbuf is such an object; it is an array of some sort. The setjmp routine then saves the state of things. When an interrupt occurs, a call is forced to the onintr routine, which can print a message, set flags, or whatever. Longjmp takes as argument an object stored into by setjmp, and restores control to the location after the call to setjmp, 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. This is necessary; most signals are automatically reset to their default action when they occur.

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 on occurrence of a signal sets a flag and then returns instead of calling exit or longjmp, execution will continue 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 terminal 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 the

terminal 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 terminal 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 reads from a terminal, wait, and pause.) A program whose onintr program 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 */
```

One item to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) 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 your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal 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 = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

The function signal obviously has a rather strange second argument. This argument is a pointer to a function delivering an integer, 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 PDP-11; the definitions are sufficiently ugly and nonportable to encourage use of the standard include file:

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

2.3 The Standard I/O Library

A knowledge of the available C libraries is essential to the C programmer, since they defines a common set of macros, types, and functions that can be used in almost any programming project. The most important functions and macros are declared in the standard I/O library, which was 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 free of the magic numbers and mysterious calls whose use can reduce understandability and portability.

3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems.

2.3.1 General Usage

Each program using the library must have the line

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

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore (_) to reduce the possibility of conflict with other names created by the user. The names intended to be visible outside the package are:

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

EOF The value returned by the read routines on end-of-file or error; usually -1

NULL The null pointer, returned by pointer-valued functions to indicate an error

FILE The name of a macro useful when declaring pointers to streams. It expands to "**struct** _iob".

BUFSIZ The size (usually 512) size suitable for an I/O buffer supplied by the user. See setbuf, below.

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, they may not have breakpoints set on them when debugging.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names stdin, stdout, and stderr are in effect constants and may not be assigned to. Stdio.h contains the definitions of NULL, EOF, FILE, and BUFSIZ. The standard input file (stdin), standard output file (stdout), and standard error file (stderr) are also defined in the standard I/O library. These definitions can be incorporated into a C program with the following statement:

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

The file ctype.h provides the macro definitions for the possible character classifications. Any program using those facilities must contain the line:

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

The functions that handle signals need to use the signal definitions, so these definitions must be included if these functions are to be used. This can be done with the line:

```
#include <signal.h>
```

Some function names have changed in order to follow the established convention. To insure that the uniqueness of function names is preserved even if truncation occurs on some systems, those functions dealing with entire strings are named str...; those functions that consider only the first n characters of a string are named strn....

Listed below are some common C library functions. Most of these belong to the standard I/O library -- although other libraries are represented here as well.

2.3.2 File Access

fclose

```
#include <stdio.h>
int fclose(stream)
FILE *stream;
```

Fclose closes a file that was opened by fopen, frees any buffers after emptying them, and returns zero on success, nonzero on error. Exit calls fclose for all open files as part of its processing.

fdopen

```
#include <stdio.h>
FILE *fdopen (fildes, type)
int fildes;
char *type;
```

Fdopen provides a bridge between the low-level input-output (I/O) facilities of XENIX and the standard I/O functions. Fdopen associates a stream with a valid file descriptor obtained from a XENIX system call (e.g., open). "Type" is the same mode ("r", "w", "a", r+ , w+ , a+) that was used in the original creation of a file identified by "fildes". Fdopen returns a pointer to the associated stream, or NULL if unsuccessful.

Example:

```
int fd;
char *name = "myfile";
FILE *strm;

fd = open(name,0);

        .
        .
        .
        if((strm = fdopen(fd,"r")) == NULL)
                fprintf(stderr,"Error on %d\n",fd);
```

fileno

```
#include <stdio.h>
int fileno (stream)
FILE *stream;
```

Implemented as a macro on XENIX, (and contained in the file stdio.h), fileno returns an integer file descriptor associated with a valid "stream". Any existing non-XENIX implementations may have different meanings for the integer which is returned. Fileno is used by many other standard functions in the C library.

fopen

```
#include <stdio.h>
FILE *fopen (filename, type)
char *filename, *type;
```

Fopen opens a file named "filename" and returns a pointer to a structure (hereafter referred to as "stream"), containing the data necessary to handle a stream of data. The "type" is one of the following character strings:

r	Used to open for reading.
w	Used to open for writing, which truncates an existing file to zero length or creates a new file.
a	Used to append, that is, open for writing at the end of a file, or create a new file.

For the update options, fseek or rewind can be used to trigger the change from reading to writing, or vice versa. (Reaching EOF on input will also permit writing without further formality.) Fopen returns a NULL pointer if "filename" cannot be opened. The update functions are particularly applicable to stream I/O and allow for the possibility of creating temporary files for both reading and writing.

Example:

```
FILE *fp;
char *file;

if((fp = fopen(file,"r")) == NULL)
        fprintf(stderr, "Cannot open %s\n",file);
```

freopen

```
#include <stdio.h>
FILE *freopen (newfile, type, stream)
char *newfile, *type;
FILE *stream;
```

Freopen accepts a pointer, "stream", to a previously opened file; the old file is closed, and then the new file is opened. The principal motivation for freopen is the desire to attach the names stdin, stdout, and stderr to specified files. On a successful freopen, the stream pointer is returned; otherwise NULL is returned, indicating that while the file closing took place, the reopening failed. Freopen is of limited portability; it cannot be implemented in all environments.

Example:

```
char *newfile;
FILE *nfile;

if((nfile = freopen(newfile,"r",stdout)) == NULL)
        fprintf(stderr,"Cannot reopen %s\n",newfile);
```

fseek

```
#include <stdio.h>
int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;
```

Fseek positions a stream to a location "offset" distance from the beginning, current position or end of a file, depending on the values 0, 1, 2 respectively for "ptrname". On XENIX the offset unit is bytes; other implementations are not necessarily the same. The return values are 0 on success and EOF on failure. Both buffered and unbuffered files may use fseek.

Example:

To position to the end of a file:

```
FILE *stream;

fseek(stream,0L,2);
```

pclose

```
#include <stdio.h>
int pclose (stream)
FILE *stream;
```

Pclose closes a stream opened by popen. It returns the exit status of the command that was issued as the first argument of its corresponding popen, or -1 if the stream was not opened by popen.

popen

```
#include <stdio.h>
FILE *popen (command, type)
char *command, *type;
```

Popen creates a pipe between the calling process and a command to be executed. The first argument is a shell command line; type is the I/O mode for the pipe, and may be either "r" for reading or "w" for writing. The function returns a stream pointer to be used for I/O on the standard input or output of the command. A NULL pointer is returned if an error occurs.

Example:

```
FILE *pstrm;

if((pstrm=popen("tr mvp MVP","w"))== NULL)
        fprintf(stderr,"popen error\n");
fprintf(pstrm,"a message via the pipe...\n");
if(pclose(pstrm) == -1)
        fprintf(stderr,"Pclose error\n");
```

results in:

```
a message via the pipe
```

rewind

```
#include <stdio.h>
int rewind(stream)
FILE *stream;
```

Rewind sets the position of the next operation at the beginning of the file associated with "stream", retaining the current mode of the file. It is the equivalent of fseek (stream,0L,0);.

setbuf

```
#include <stdio.h>
setbuf (stream, buf)
FILE *stream;
char *buf;
```

This function allows the user to choose his own buffer for I/O or choose no buffering at all. Use it after opening and before reading or writing. The function is often used to eliminate the single character writes to a file that result from the execution of putc to standard output that is not redirected. The choice to buffer I/O brings with it the responsibility for flushing any data that may remain in a last, partially-filled buffer. Fflush or fclose perform this task. The constant BUFSIZ in stdio.h tells how big the character array "buf" is. It is well-chosen for the machine on which XENIX is running. When "buf" is set to NULL, the I/O is completely unbuffered.

Example:

```
setbuf (stdout, malloc(BUFSIZ));
```

2.3.3 File Status

clearerr

```
#include <stdio.h>
clearerr(stream)
FILE *stream;
```

Clearerr resets the error condition on "stream". The need for clearerr arises in XENIX implementations where the error indicator is not reset after a query.

feof

```
#include <stdio.h>
int feof (stream)
FILE *stream;
```

Feof, which is implemented as a macro, returns nonzero if an input operation on "stream" has reached end of file; otherwise a zero is returned. Feof should be used in conjunction with any I/O function whose return value is not a clear indicator of an end-of-file condition. Such functions are fread and getw.

Example:

```
int *x;
FILE *stream;

do
        *x++ = getw(stream);
while(!feof(stream));
```

ferror

```
#include <stdio.h>
int ferror (stream)
FILE *stream;
```

Ferror tests for an indication of error on "stream". It returns a nonzero value (true) when an error is found, and a zero otherwise. Calls to ferror do not clear the error condition, hence the clearerr function is needed for that purpose. The user should be aware that, after an error, further use of the file may cause strange results. On XENIX ferror is implemented as a macro.

Example:

```
FILE *stream;
int *x;

while(!ferror(stream))
        putw(*x++,stream);
```

ftell

```
#include <stdio.h>
long ftell (stream)
FILE *stream;
```

Ftell determines the current offset relative to the beginning of the file associated with "stream". It returns the current value of the offset in bytes. On error, a value of -1 is returned. This function is useful in obtaining an offset for subsequent fseek calls.

2.3.4 Input Function

fgetc

```
#include <stdio.h>
int fgetc (stream)
FILE *stream;
```

This is the function version of the macro getc and acts identically to getc. Because fgetc is a function and not a macro, it can be used in debugging to set breakpoints on fgetc and when the side effects of macro processing of the argument is a problem. Furthermore, it can also be passed as an argument.

fgets

```
#include <stdio.h>
char *fgets (s,n,stream)
char *s;
int n;
FILE *stream;
```

Fgets reads from "stream" into the area pointed to by "s" either n-1 characters or an entire string including its newline terminator, whichever comes first. A final null character is affixed to the data read. Fgets returns the pointer "s" on success, and NULL on end-of-file or error. Fgets differs from the function gets in three ways: it can read from other than stdin; it appends the newline at the end of input when the size of the string is longer than or equal to "n"; and even more important, it provides control, not available with gets, over the size of the string to be read.

Example:

```
char msg[MAX];
FILE *myfile;

        while(fgets(msg,MAX,myfile) != NULL)
                printf("%s\n",msg);
```

fread

```
#include <stdio.h>
int fread((char *)ptr, sizeof (*ptr), nitems, stream)
FILE *stream;
```

This function reads from "stream" the next "nitems" whose size is the same as the size of the item pointed to by "ptr", into a sufficiently large area starting at "ptr". It returns the number of items read. In XENIX, fread makes use of the function getc. It is often used in combination with feof and ferror to obtain a clear indication of the file status.

Example:

```
FILE *pstm;
char mesg[100];

while(fread((char *)mesg,sizeof(*mesg),1,pstm) == 1)
        printf("%s\n",mesg);
```

fscanf

```
#include <stdio.h>
int fscanf (stream, format[, argptr]...)
char *format;
FILE *stream;
```

Fscanf accepts input from the file associated with "stream", and deposits it into the storage area pointed to by the respective argument pointers according to the specified formats. Fscanf differs from scanf in that it can read from other than stdin. The function returns the number of successfully handled input arguments, or EOF on end of input.

Example:

```
FILE *file;
long pay;
char name[15];
char pan[7];

fscanf(file,"%6s%14s%ld\n",pan,name,&pay);
if(pay<50000)
        printf("$%ld raise for %s.\n",pay/10,name);
```

If the input data is:

```
020202MaryJones 15000
```

the resulting output is:

$1500 raise for MaryJones.

getc

```
#include <stdio.h>
int getc (stream)
FILE *stream;
```

Getc returns the next character from the named "stream". It is implemented as a macro to avoid the overhead of a function call. On error or end-of-file it returns an EOF. Fgetc should be used if it is necessary to avoid the side effects of argument processing by the macro getc.

getchar

```
#include <stdio.h>
int getchar()
```

This is identical to getc (stdin).

gets

```
#include <stdio.h>
char *gets(s)
char *s;
```

Gets reads a string of characters up to a newline from stdin and places them in the area pointed to by "s". The newline character which ended the string is replaced by the null character. The return values are "s" on success, NULL on error or end-of-file. The simple example below presumes the size of the string read into "msg" will not exceed SIZE in length. If used in conjunction with strlen, a dangerous overflow can be detected, though not prevented.

Example:

```
char msg[SIZE];
char *s;
        s = msg;
        while (gets(s) != NULL)
                printf("%s\n",s);
```

getw

```
#include <stdio.h>
int getw (stream)
FILE *stream;
```

Getw reads the next word from the file associated with "stream". If successful, it returns the word; on error or end-of-file, it returns EOF. However, because EOF could be a valid word, this function is best used with feof and ferror.

Example:

```
FILE *stream;
int *x;
        do
                *x++ = getw(stream);
        while (!feof(stream));
```

scanf

```
#include <stdio.h>
int scanf (format[, argptr]...)
char *format;
```

Scanf reads input from stdin, delivers the input according to the specified formats, and deposits the input in the storage area pointed to by the respective argument pointers. For input from other streams than stdin use fscanf; for input from a character array use sscanf. Scanf returns the number of successfully handled input arguments, or EOF on end-of-input.

Example:

```
long number;

scanf("%ld",&number);
(printf(number%2?"%ld is odd":"%ld is even",number));
```

sscanf

```
#include <stdio.h>
sscanf (s, format [, pointer]...)
char *s;
char *format;
```

Sscanf accepts input from character string "s", delivers the input according to the specified formats, and deposits it into the storage area pointed to by the respective argument pointers. This function returns the number of successfully handled input arguments.

Example:

```
char datestr[] = {"THU MAR 29 11:04:40 EST 1983"};
char month[4];
char year[5];

sscanf(datestr,"%*3s%3s%*2s%*8s%*3s%4s",month,year);
printf("%s, %s\n",month,year);
```

The result is:

```
MAR, 1983
```

ungetc

```
#include <stdio.h>
int ungetc (c, stream)
int c;
FILE *stream;
```

Ungetc puts the character "c" back on the file associated with "stream". One character (but never EOF) is assured of being put back. If successful, the function returns "c", otherwise EOF.

Example:

```
while(isspace (c = getc(stdin)))
        ;
ungetc(c,stdin);
```

This code puts the first character that is not white space back onto the standard input stream.

2.3.5 Output Functions

fflush

```
#include <stdio.h>
int fflush (stream)
FILE *stream;
```

Fflush takes action to guarantee that any data contained in file buffers and not yet written out will be written. It is used by fclose to flush a stream. No action is taken on files not open for writing. The return values are zero for success, EOF on error.

fprintf

```
#include <stdio.h>
int fprintf (stream, format[, arg ]...)
FILE *stream;
char *format;
```

Fprintf provides formatted output to a named stream. The function printf may be used if the destination is stdout. Fprintf returns nonzero on error, otherwise zero.

Example:

```
int *filename;
int c;

if(c==EOF)
        fprintf(stderr,"EOF on %s\n",filename);
```

fputc

```
#include <stdio.h>
int fputc (c,stream)
int c;
FILE *stream;
```

Fputc performs the same task as putc; that is, it writes the character "c" to the file associated with "stream", but is implemented as a function rather than a macro. Fputc is preferred to putc when the side effects of macro processing of arguments are a problem. On success, it returns the character written; on failure it returns EOF.

Example:

```
FILE *in, *out;
int c;

        while ((c = fgetc(in)) != EOF)
                fputc(c,out);
```

fputs

```
#include <stdio.h>
int fputs(s,stream)
char *s;
FILE *stream;
```

Fputs copies a string to the output file associated with "stream", using the function putc

to do this. It is different from puts in two ways: fputs allows any output stream to be specified, and does not affix a newline to the output. For an example, see puts.

fwrite

```
#include <stdio.h>
int fwrite ((char *)ptr, sizeof (*ptr),nitems,stream)
FILE *stream;
```

Beginning at "ptr", this function writes up to "nitems" of data of the type pointed to by "ptr" into output "stream". It returns the number of items actually written. Like fread, this function should be used in conjunction with ferror to detect the error condition.

Example:

```
char mesg[] ={"My message to write out\n"};
FILE *pstrm;

if(fwrite(mesg,(sizeof(*mesg)-1),1,pstrm) != 1)
        fprintf(stderr,"Output error\n");
```

printf

```
#include <stdio.h>
int printf(format[, arg]...)
char *format;
```

Printf provides formatted output on stdout. Fprintf and sprintf are related functions that write output onto other than the standard output device. In case of error, implementations are not consistent in their output. On error, printf returns nonzero, otherwise zero. In later releases of the C library, printf returns the number of characters transmitted, or a negative value on error.

Example:

```
int num = 10;
char msg[] = {"ten"};
printf("%d - %o - %s\n", num, num, msg);
```

results in the line:

```
10 - 12 - ten;
```

putc

```
#include <stdio.h>
int putc (c,stream)
int c;
FILE *stream;
```

Putc writes the character c to the file associated with stream. On success, it returns the character written; on error it returns EOF. Because it is implemented as a macro, side effects may result from argument processing. In such cases, the equivalent function fputc should be used.

Example:

```
#define PROMPT()        putc('\7',stderr)
/* Prompt is BELL character */
```

putchar

```
#include <stdio.h>
int putchar(c)
int c;
```

Putchar is defined as:

```
putc (c, stdout)
```

Putchar returns the character written or EOF if an error occurs.

Example:

```
char *cp;
char x[SIZE];

        for(cp=x;cp<(x+SIZE);cp++)
                putchar(*cp);
```

puts

```
#include <stdio.h>
int puts(s)
char *s;
```

The function puts copies the string pointed to by "s" without its terminating null character to stdout. A newline character is appended. XENIX

uses the macro putchar (which calls putc).

Example:

```
puts("I will append a newline");
fputs("\tsome more data ", stdout);
puts("and now a newline");
```

The resulting output is:

```
I will append a newline
        some more data and now a newline
```

putw

```
#include <stdio.h>
int putw(w,stream)
FILE *stream;
int w;
```

Putw appends word "w" to the output "stream". As with getw, the proper way to check for an error or end-of-file is to use the feof and ferror functions.

Example:

```
int info;

while(!feof(stream))
        putw(info,stream);
```

sprintf

```
#include <stdio.h>
int sprintf(s, format, [, arg]...)
char *s;
char *format;
```

Sprintf allows formatted output to be placed in a character array pointed to by "s". Sprintf adds a null at the end of the formatted output. It is the user's responsibility to provide an array of sufficient length. The related functions printf and fprintf handle similar kinds of formatted output. The comparable input function is sscanf. On error, sprintf returns nonzero, otherwise zero.

Example:

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c"
int width = 50;
int length = 60;

sprintf(cmd,"pr -w%d -l%d %s\n",width,length,doc);
system(cmd);
```

The above code executes the pr command to print the source of the cp command.

2.3.6 String Functions

strcat

```
char *strcat(dst,src)
char *dst, *src;
```

Strcat appends characters in the string pointed to by "src" to the end of the string pointed to by "dst", and places a null character after the last character copied. It returns a pointer to "dst". To concatenate strings up to a maximum number of characters, use strncat.

Example:

```
char *myfile;
char dir[L_cuserid+5] = "/usr/";
        myfile = (strcat(dir,cuserid(0)));
```

The result is the concatenation of the login name onto the end of the string "dir".

strcmp

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

Strcmp compares the characters in the string "s1" and "s2". It returns an integer value, greater than, equal to, or less than zero, depending on whether "s1" is lexicographically greater than, equal to, or less than "s2".

Example:

```
#define EQ(x,y)  !strcmp(x,y)
```

strcpy

```
char *strcpy(dst, src)
char *dst, *src;
```

Strcpy copies the characters (including the null terminator) from the string pointed to by "src" into the string pointed to by "dst". A pointer to "dst" is returned.

Example:

```
char dst[] = "UPPER CASE";
char src[] = "this is lowercase";

        printf("%s\n",strcpy(dst,src+8));
```

results in:

```
lowercase
```

strlen

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

Strlen counts the number of characters starting at the character pointed to by "s" up to, but not including, the first null character. It returns the integer count.

Example:

```
char nextitem[SIZE];
char series[MAX];

if(strlen(series)) strcat(series,",");
strcat(series,nextitem);
```

strncat

```
char *strncat(dst, src, n)
char *dst, *src;
int n;
```

Strncat appends a maximum of "n" characters of the string pointed to by "src" and then a null

character to the string pointed to by "dst". It returns a pointer to "dst".

Example:

```
char dst[] = "cover";
char src[] = "letter";

        printf("%s\n",strncat(dst,src,3));
```

The output is:

```
coverlet
```

strncmp

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

Strncmp compares two strings for at most "n" characters and returns an integer value greater than, equal to, or less than zero depending on whether "s1" is lexicographically greater than, equal to or less than "s2".

Example:

```
char filename [] = "/dev/ttyx";

if(strncmp (filename+5, "tty",3) == 0)
        printf("success\n");
```

strncpy

```
char *strncpy(dst,src,n)
char *dst, *src;
int n;
```

Strncpy copies "n" characters of the string pointed to by "src" into the string pointed to by "dst". Null padding or truncation of "src" occurs as necessary. A pointer to "dst" is returned.

Example:

```
char buf [MAX];
char date [29] = {"Fri Dec 29 09:35:44 EDT 1982"};
char *day = buf;

        strncpy(day,date,3);
```

After executing this code, "day" points to the string "Fri".

2.3.7 Character Classification

isalnum

```
#include <ctype.h>
int isalnum(c)
int c;
```

This macro determines whether or not the character "c" is an alphanumeric character ([A-Za-z0-9]). It returns zero for false and nonzero for true.

isalpha

```
#include <ctype.h>
int isalpha(c)
int c;
```

This macro determines whether or not the character "c" is an alphabetic character ([A-Za-z]). It returns zero for false and nonzero for true.

isascii

```
#include <ctype.h>
int isascii(c)
int c;
```

This macro determines whether or not the integer value supplied is an ASCII character; that is, a character whose octal value ranges from 000 to 177. It returns zero for false and nonzero for true.

iscntrl

```
#include <ctype.h>
int iscntrl(c)
int c;
```

This macro determines whether or not the character "c" when mapped to ASCII is a control

character (that is, octal 177 or 000-037). It returns zero for false and nonzero for true.

isdigit

```
#include <ctype.h>
int isdigit(c)
int c;
```

This macro determines whether or not the character "c" is a digit. It returns zero for false and nonzero for true. (that is, is an ASCII code between octal 041 and 176 inclusive).

islower

```
#include <ctype.h>
int islower(c)
int c;
```

This macro determines whether or not the character "c" is a lowercase letter. It returns zero for false and nonzero for true.

isprint

```
#include <ctype.h>
int isprint(c)
int c;
```

This macro determines whether or not the character "c" is a printable character. (This includes spaces.) It returns zero for false and nonzero for true.

ispunct

```
#include <ctype.h>
int ispunct(c)
int c;
```

This macro determines whether or not the character "c" is a punctuation character (neither a control character nor an alphanumeric). It returns zero for false and nonzero for true.

isspace

```
#include <ctype.h>
int isspace(c)
int c;
```

This macro determines whether or not the character "c" is a form of white space (that is, a blank, horizontal or vertical tab, carriage return, form-feed or newline). It returns zero

for false and nonzero for true.

isupper

```
#include <ctype.h>
int isupper(c)
int c;
```

This macro determines whether or not the character "c" is an uppercase letter. It returns zero for false and nonzero for true.

2.3.8 Character Translation

toascii

```
#include <ctype.h>
int toascii (c)
int c;
```

The macro toascii usually does nothing: its purpose is to map the input character into its ASCII equivalent.

Example:

```
FILE *oddstrm;

        if(!isdigit (toascii(getw(oddstrm))))
                fprintf(stderr,"bad data\n");
```

tolower

```
#include <ctype.h>
int tolower (c)
int c;
```

If the argument "c" passed to the function tolower is an uppercase letter, the lowercase representation of "c" is returned, otherwise "c" is returned unchanged. For a faster routine, use tolower, which is implemented as a macro; however, the argument must already be an uppercase letter.

Example:

```
if(tolower(getchar()) != 'y')
        exit(0);
```

toupper

```
#include <ctype.h>
int toupper (c)
int c;
```

If the argument "c" passed to the function toupper is a lowercase letter, the uppercase representation of "c" is returned, otherwise "c" is returned unchanged. For a faster routine, use toupper, however, the argument must already be a lowercase letter.

Example:

```
if(toupper (getchar()) != 'Y')
        exit(0);
```

2.3.9 Space Allocation

calloc

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

Calloc allocates enough storage for an array of "n" items aligned for any use, each of "size" bytes. The space is initialized to zero. Calloc returns a pointer to the beginning of the allocated space, or a NULL pointer on failure.

Example:

```
char *t;
int n;
unsigned size;

        if(t=calloc((unsigned)n, size) == NULL)
                fprintf(stderr,"Out of space.\n");
```

free

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

Free is used in conjunction with the space allocating functions malloc, calloc, or realloc. "Ptr" is a pointer supplied by one of these routines. The function frees the space previously allocated.

malloc

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

Malloc allocates "size" bytes of storage beginning on a word boundary. It returns a pointer to the beginning of the allocated space, or a NULL pointer on failure to acquire space. For space initialized to zero, see calloc.

Example:

```
int n;
char *t;
unsigned size;

    if(t=malloc((unsigned)n) == NULL)
        fprintf(stderr,"Out of space.\n");
```

realloc

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

Given "ptr" which was supplied by a call to malloc or calloc, and a new byte size, "size", realloc returns a pointer to the block of space of "size" bytes. This function is used to compact storage, and is used with the functions malloc and free.

2.4 Include Files

The following pages contain the contents of the three most important include files: ctype.h, stdio.h, and signal.s. These files are well worth some study, as the define a standard interface to the internals of the XENIX system.

2.4.1 ctype.h

```
#define _U      01
#define _L      02
#define _N      04
#define _S     010
#define _P     020
#define _C     040
#define _B    0100

extern  char    _ctype_[];

#define isalpha(c)      ((_ctype_+1)[c]&(_U|_L))
#define isupper(c)      ((_ctype_+1)[c]&_U)
#define islower(c)      ((_ctype_+1)[c]&_L)
#define isdigit(c)      ((_ctype_+1)[c]&_N)
#define isspace(c)      ((_ctype_+1)[c]&(_S|_B))
#define ispunct(c)      ((_ctype_+1)[c]&_P)
#define isalnum(c)      ((_ctype_+1)[c]&(_U|_L|_N))
#define isprint(c)      ((_ctype_+1)[c]&(_P|_U|_L|_N|_B))
#define iscntrl(c)      ((_ctype_+1)[c]&_C)
#define isascii(c)      ((unsigned)(c)<=0177)
#define _toupper(c)     ((c)-'a'+'A')
#define _tolower(c)     ((c)-'A'+'a')
#define toascii(c)      ((c)&0177)
```

2.4.2 signal.h

```
#define     NSIG        16

#define     SIGHUP       1   /* hangup */
#define     SIGINT       2   /* interrupt */
#define     SIGQUIT      3   /* quit */
#define     SIGILL       4   /* illegal instruction */
                             /* (not reset when caught) */
#define     SIGTRAP      5   /* trace trap */
                             /* (not reset when caught) */
#define     SIGIOT       6   /* IOT instruction */
#define     SIGEMT       7   /* EMT instruction */
#define     SIGFPE       8   /* floating point exception */
#define     SIGKILL      9   /* kill (cannot be */
                             /* (caught or ignored) */
#define     SIGBUS      10   /* bus error */
#define     SIGSEGV     11   /* segmentation violation */
#define     SIGSYS      12   /* bad argument to system call */
#define     SIGPIPE     13   /* write on a pipe */
                             /* with no one to read it */
#define     SIGALRM     14   /* alarm clock */
#define     SIGTERM     15   /* software termination */
                             /* signal from kill */

int         (*signal())();
#define     SIG_DFL     (int (*)())0
#define     SIG_IGN     (int (*)())1
```

2.4.3 stdio.h

```
#define         BUFSIZ          512
#define         _NFILE           20
# ifndef FILE
extern  struct  _iobuf {
        char    *_ptr;
        int     _cnt;
        char    *_base;
        char    _flag;
        char    _file;
} _iob[_NFILE];
# endif

#define         _IOREAD          01
#define         _IOWRT           02
#define         _IONBF           04
#define         _IOMYBUF        010
#define         _IOEOF          020
#define         _IOERR          040
#define         _IOSTRG        0100
#define         _IORW          0200

#define         NULL             0
#define         FILE            struct  _iobuf
#define         EOF             (-1)

#define         L_ctermid        9
#define         L_cuserid        9
#define         L_tmpnam        19

#define         stdin           (&_iob[0])
#define         stdout          (&_iob[1])
#define         stderr          (&_iob[2])
#define         getc(p)         (--(p)->_cnt>=0?\
                                *(p)->_ptr++&0377:_filbuf(p))
#define         getchar()       getc(stdin)
#define         putc(x,p)       (--(p)->_cnt>=0?\
                                ((int)(*(p)->_ptr++=(unsigned)(x))):\
                                _flsbuf((unsigned)(x),p))
#define         putchar(x)      putc(x,stdout)
#define         feof(p)         (((p)->_flag&_IOEOF)!=0)
#define         ferror(p)       (((p)->_flag&_IOERR)!=0)
#define         fileno(p)       p->_file

FILE    *fopen();
FILE    *freopen();
FILE    *fdopen();
long    ftell();
char    *fgets();
```

2.5 XENIX MC68000 Assembly Language Interface

The XENIX system is designed so that there should be little need to program in assembly language. Occasionally, however, the need does arise, and you may need to know the conventions for storing words in memory and for accessing parameters on the stack in a way compatible with the C runtime environment. Remember, however, that programming in assembly language is highly machine-dependent, and that you sacrifice portability whenever you forsake C for whatever low-level advantages you might gain.

If you do choose to mix MC68000 assembly language routines and compiled C routines, there are several things to be aware of:

- Registers and return values
- Calling sequence
- Stack probes

With an understanding of these three topics, you should be able to write both C programs that call MC68000 assembly language routines and assembly language routines that call compiled C routines.

2.5.1 Registers and Return Values

Function return values are passed in registers if possible. The set of machine registers used is called the save set, and includes the registers from d2-d7 and a2-a7 that are modified by a routine. The compiler assumes that these registers are preserved by the callee, and saves them itself when it is generating code for the callee. (When a C compatible routine is called by another routine, we refer to the calling routine as the caller. We refer to the called routine as the callee.) Note that a6 and a7 are in effect saved by a link instruction at procedure entry.

The function return value is in d0. The current floating point implementation returns the high order 32 bits of doubles in d1, and the low order 32 bits in d0. Functions return structure values (not pointers to the values) by loading d0 with a pointer to a static buffer containing the

structure value.

This makes the following two functions equivalent:

```
struct list proc (){
        struct list this;
        ...
        return (this);
}

struct list *proc (){
        struct list this;
        static struct list temp;
        ...
        temp = this;
        return (&temp);
}
```

This implementation allows recursive reentrancy (as long as the explicit form is not used, since the first sequence is indivisible but not the second). However, this implementation does not permit multitasking reentrancy. Note that the latter includes the XENIX signal(3) call.

Setjmp(3) and longjmp(3) cannot be implemented as they are on the PDP-11, because each procedure saves only the registers from the save set that it will modify. This makes it difficult to get back the current values of the register variables of the procedure that is being setjmped to. Hence, register variable values after a longjmp are the same as before a corresponding setjmp is called. If you need local variables to change between the call of setjmp and longjmp, they cannot be register variables.

2.5.2 Calling Sequence

The calling sequence is straightforward: arguments are pushed on the stack from the last to first: i.e., from right to left as you read them in the C source. The push quantum is 4 bytes, so if you are pushing a character, you must extend it appropriately before pushing. Structures and floating point numbers that are larger than 4 bytes are pushed in increments of 4 bytes so that they end up in the same order in stack memory as they are in any other memory. This means pushing the last word first and long-word padding the last word (the first pushed) if necessary. The caller is responsible for removing his own arguments. Typically, an

```
addql   #constant,sp
```

is done. It is not really important whether the caller actually pushes and pops his arguments or just stores them in a static area at the top of the stack, but the debugger, adb, examines the **addql** or **addw** from the sp to decide how many arguments there were.

2.5.3 Stack Probes

XENIX is designed to dynamically allocate space on the stack for local variables, function arguments, return addresses, and other information. When additional space is needed and an instruction causes a memory fault, the XENIX kernel checks the offending instruction. If the instruction is a stack reference, the kernel maps enough stack memory for the instruction to complete its execution successfully. Then the procedure continues execution where it left off. Generally, this means restarting the offending memory reference instruction (usually a push or store). The MC68000 does not provide a way to restart instructions; therefore, we need to perform a special instruction that can trigger the memory fault, but that has no ill effect other than triggering the fault. This instruction we call a stack probe.

When we perform a stack probe and a memory fault occurs, the kernel allocates additional memory for the stack. The XENIX kernel can allocate this needed stack memory, ignore the fact that the stack probe instruction did not complete, and then continue on to the next instruction.

The stack probe instruction for the MC68000 XENIX is

```
tstb -value(sp)
```

The value argument must be negative, because a negative index from the stack pointer is above the top of the stack. This is an otherwise absurd reference that XENIX recognizes as a stack probe.

For the general case, use the following procedure entry sequence:

```
procedure_entry:
        link    a6,#-savesize
        tstb    -pushsize-extra-8(sp)
```

Any registers among d2-d7 and a2-a5 that are used in this procedure are saved with a **moveml** instruction after this sequence. The number of registers saved in the **moveml** instruction needs to be accounted for in the push size. Thus, pushsize is the sum of the number of bytes pushed as

temporaries, save areas, and arguments by the whole procedure. The 8 bytes are the space for the return address and frame pointer save (by the link instruction) of a nested call. The extra is tolerance so that extremely short runtimes that use little stack do not need to perform a stack probe. The tolerance is intentionally kept small to conserve memory, so make sure you understand what you are doing before you consider leaving out a stack probe in your assembly procedures.

In most cases, unless you are pushing huge structures or doing tricks with the stack within your procedure, you can use the following instruction for your stack probe:

```
tstb    -100(sp)
```

This instruction makes sure that enough space has been allocated for most of the usual things you might do with the stack and is enough for the XENIX runtimes that do not perform stack probes. Note that you do not need to consider space allocated by the link instruction in this stack probe, since it is already added by indexing off the stack pointer.

CHAPTER 3

SOFTWARE TOOLS

CONTENTS

3.1 Introduction

This chapter discusses the tools available for use in software development on the XENIX System. The wide variety of available tools makes for a rich environment for programmers and software engineers. Often, tools are combined in shell procedures to perform whatever programming task desired. However, because their are so many different commands available to the programmer, it is often difficult to know what subset is especially useful in software development. To solve this problem, this chapter circumscribes a set of commands that are known to be of use in software development, and then summarizes the function of each command. In addition, this chapter contains a section describing the basic tools required in developing software on the XENIX system. Most of these basic tools have late chapters devoted to them.

3.2 Basic Tools

The tools used to create executable C programs are:

cc The XENIX C compiler.

lint The XENIX C program checker.

ld The XENIX loader.

as The XENIX assembler.

adb The XENIX debugger.

make The XENIX program maintainer.

Note that **cc** automatically invokes both the loader and the assembler so that use of either is optional. **Lint** is normally used in the early stages of program development to check for illegal and improper usage of the C language. The program **adb** is used to debug executable programs. The program **make** is used with the above tools to automatically maintain and regenerate software in medium scale programming projects.

All the above tools are used in creating executable C programs. These programs are created to run in the XENIX environment. This environment is manifested in the various subroutines and system calls available in several subroutine libraries.

Note that not all programming projects are best implemented in C, even if they are programs written for XENIX. Often, simple programs can be written in the shell command language much more quickly than they can be in C. For some complicated programs, **lex** and yacc may be just what is required. **Lex** is a lexical analyzer that can be used to accept a given input language. **Yacc** is a program designed to compile grammars into a parsing program. Typically, it is used to compile languages that are recognized by **lex**. For this reason, **lex** and **yacc** are often used together, although either can be used separately.

3.3 Other Tools

Other tools useful in software development are described below:

ar Archives files and maintains libraries. Useful when constructing or maintaining large object libraries.

at Execute commands at a later time. Used to execute time consuming compilations, printouts, and **makes**, so that they execute when the system isn't busy.

awk Recognizes text patterns and performs transformation operations based on the awk language.

basename Strips directory affixes and filename prefixes from a filename or pathname. Useful in shell scripts to obtain the filename part of a pathname or to strip off filename extensions.

cb Beautifies C programs, improving their readability. Note that the output from cb is not necessarily attractive to all programmers.

chgrp Changes the group affiliation of a file, so that it has the proper group permission when it is accessed or, if it is an executable file, when it is executed.

chown Changes the owner of a file, so that it has the proper owner when it is accessed or, if it is an executable file, when it is executed.

cmp Compares two sorted files. Useful when comparing object files and other binary images.

comm	Compares sorted lists by either selecting or rejecting common lines.
copy	Copies groups of files. Useful in recursively copying directories or in creating files that are linked to another set of files. Note that cp does not copy recursively.
csh	Interprets and executes commands taken from the keyboard or from a command file. The "C shell" supports a C-like command language, an aliasing facility, and a command history mechanism.
ctags	Creates a tags file so that C functions can be quickly found in a set of related C source files.
dd	Converts and copies a file to the standard output. Used to read in files from various media. A variety of formats and conversions are available.
df	Reports the amount of space that is free and available in a given file system.
diff	Compares two text files, line by line.
diff3	Compares three text files, line by line.
du	Summarizes disk usage. Used to determine how much disk space you are using.
file	Determines the file type of given files Use this to examine files to determine whether they are directories, special files, or executable files.
find	Finds filenames in a filesystem and optionally may perform commands that affect the found files. Used in performing complex operations on a selected set of files.
fsck	Checks file system consistency and if possible, interactively repairs the file system when inconsistencies are found.
ln	Creates a link of a file to another file so that file contents are shared and both filenames refer to the same file.
lorder	Finds ordering relation for an object library.

m4	Processes input text performing several functions, including macro definition and invocation.
mkfs	Constructs a file system.
mknod	Creates a special file.
mkstr	Creates an error message file by examining a C source file.
mount	Mounts a file system on the given directory.
ncheck	Generate file names from inode numbers.
nice	Runs a command at a lower priority.
nm	Prints the list of names in a program.
od	Performs an "octal dump" of given files, printing files in a variety of formats, one of which is octal.
printenv	Prints out the environment.
prof	Displays profile data.
pstat	Prints system facts.
quot	Summarizes file system ownership.
ranlib	Converts archive libraries to random libraries by placing a table of contents at the front of each library.
settime	Change the access and modification dates of files.
size	Reports the size of an object file.
sleep	Suspends execution for a given period of time.
strings	Finds and prints readable text (strings) in an object or other binary file.
strip	Removes symbols and relocation bits from executable files.
su	Logs in user as super-user or other user.

sum	Computes check sum for a file and counts blocks. Useful in looking for bad spots in a file and in verifying transmission of data between systems.
sync	Updates the super block so that all input and output to the disk is completed before the **sync** command finishes.
tar	Archives files to tape or other similar device. Also used in moving large sets of files.
time	Times a given command. Used in taking benchmarks for execution-time of programs.
touch	Updates the modification date of a file without altering the contents of the file.
tr	Translates a one given set of characters to another set for all characters in a file.
tsort	Topologically sorts object libraries so that dependencies are apparent.
umount	Unmounts a mounted file system.
xstr	Extracts strings from C programs to implement shared strings.

CHAPTER 4

CC: A C COMPILER

CONTENTS

4.1 Introduction

Cc is the command used to invoke the XENIX C compiler. Since the entire XENIX system is written in the C language, cc is the fundamental XENIX program development tool. The emphasis in this chapter is on giving insight into cc's operation and use. Special emphasis is given to input and output files and and to the available compiler options. Throughout, familiarity with compilers and with the C language is assumed. For more information on programming in C, see The C Programming Language, by Kernighan and Ritchie.

The fundamental function of the C compiler is to produce executable programs by processing C source files. The word "processing" is the key here, since the compilation process involves several distinct phases: These phases are described below:

Preprocessing

In this phase of compilation, your C source program is examined for macro definitions and include file directives. Any include files are processed at the point of the include statement; then occurrences of macros are expanded throughout the text. Normally, standard include files found in the /usr/include directory are included at the beginning of C programs. These standard include files normally are named with a ".h" extension. For example, the following statement includes the definitions for functions in the standard I/O library:

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

Note that the angle brackets indicate that the file is presumed to exist in /usr/include. The effects of preprocessing on a file can be captured in a file by specifying the -P switch on the cc command line. The -E switch performs a similar function useful for debugging when you suspect that an include file or macro is not expanding as desired.

Optimization

Optimization of generated code can be specified on the cc command line with the -O switch. This option should be used to increase execution speed or to decrease size of the executing program. Since programs will take longer to compile with this option, you may want to use this option only after you have a working debugged program.

Generation of Assembly Source Code

The C compiler generates assembly source code that is later assembled by the XENIX assembler, **as**. Cc's assembly output can be saved in a file by specifying the -S switch when the compiler is invoked. Assembly source output is saved in a file whose name has the ".s" extension.

Assembly

To assemble the generated assembly code, **cc** calls as to create a ".o" file. The ".o" file is used in the next step, linking and loading.

Linking and Loading

The final phase in the compilation of a C program is linking and loading. The program responsible for this is the XENIX loader, **ld**. Loader options can be specified on the **cc** command line. These options are discussed later in the section on the loader.

It is important to realize that all of the above phases can be controlled at the **cc** command level: they do not have to be invoked separately. What normally happens when you execute a cc command is that a sequence of programs processes the original C source file. Each program creates a temporary file that is used by the next program in the sequence. The final output is the load image that is loaded into memory when the final executable file is run.

4.2 Invocation Switches

A list of some of the available switches follows. Other switches may be described in cc(1S).

-c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-p Arrange for the compiler to produce code which counts the number of times each routine is called. Also, if loading takes place, replace the standard startup routine by one that automatically calls monitor(3) at the start and arranges to write out a mon.out file at normal termination of execution of the object program. An execution profile can then be generated by use of prof(1).

-O Invoke an object-code optimizer.

-S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed ".s".

-P Run only the macro preprocessor and place the result for each ".c" file in a corresponding ".i" file. The resultant file has no "#" lines in it.

-o output
Give the final output file the name specified by output. If this option is used the file a.out will be left undisturbed.

-Dname=def
Define the name to the compiler preprocessor, as if by "#define". If no definition is given, then name is assigned the value 1.

-Uname
Remove any initial definition of name.

-Idir
Any "#include" files whose names do not begin with "/" and that are enclosed within angle brackets (< and >) are searched for first in the directory of the file argument, then in directories named in -I options, then in directories given by a standard list.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier cc run, or perhaps libraries of C-compatible routines created with the assembler. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with the name a.out.

Note that some versions of the C compiler support additional switches. These switches and their function are described in the reference section of this manual.

4.3 The Loader

As mentioned in the above sections, the XENIX loader, ld, plays a fundamental role in the development of any C program. For this reason it is discussed as part of cc; it can however, be used as a stand-alone processor of object files. Note that arguments to ld can be given on the cc

command line and are a part of the syntax of the **cc** command.

Some of the available loader switches are listed below. Except for **-l**, they should appear before filename arguments. Other switches are described in ld(1S).

-s "Strip" the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by strip(1S).

-u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-lx This option is an abbreviation for the library name /lib/libx.a, where x is a string. If that does not exist, ld tries /usr/lib/libx.a. A library is searched when its name is encountered, so the placement of a **-l** is significant.

-x Do not preserve local (non-.globl) symbols in the output symbol table; enter only external symbols. This option saves some space in the output file.

-X Save local symbols except for those whose names begin with "L". This option is used by cc to discard internally generated labels while retaining symbols local to routines.

-n Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file.

-i When the output file is executed, the program text and data areas will live in separate address spaces. The only difference between this option and **-n** is that here the data starts at location 0.

-o The name argument after **-o** is used as the name of the ld output file, instead of a.out.

For more information on the loader, see **ld** in the reference section of this manual.

4.4 Files

The files making up the compiler, as well as those files needed, used, or created by cc are listed below:

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporaries for cc
/lib/cpp	preprocessor
/lib/c[01]	compiler for cc
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard library
/usr/include	standard directory for "#include" files

CHAPTER 5

LINT: A C PROGRAM CHECKER

CONTENTS

5.1 Introduction

Lint is a program that examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

The separation of function between Lint and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not perform sophisticated type checking, especially between separately compiled programs. Lint takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This section discusses the use of Lint, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

Suppose there are two C source files, file1.c and file2.c, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

produces, in addition to the above messages, additional messages that relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** produces messages about various error-prone or wasteful constructions that, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the discussion of Lint closes with sections discussing the implementation and giving suggestions for writing portable C. The final section gives a summary of Lint command line options.

5.2 A Word About Philosophy

Many of the facts about a particular C program that Lint needs may be impossible for it to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether exit is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the Lint algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, Lint assumes it can be called: this is not necessarily so, but in practice it is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form "xxx might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages that Lint produces.

5.3 Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs. If a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions that are defined but not otherwise mentioned. An exception is made for variables that are declared through explicit **extern** statements but are never referenced. Thus, the statement

```
extern float sin();
```

will evoke no comment if sin is never used. This agrees with the semantics of the C compiler.

In some cases, these unused external declarations might be of some interest: they can be discovered by adding the -x flag to the Lint invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The -v option is available to suppress the printing of complaints about unused arguments. When -v is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when Lint is applied to some, but not all, files out of a collection that are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The -u flag may be used to suppress the spurious messages that might otherwise appear.

5.4 Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well: many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. Lint detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that Lint can complain about some legal programs, but these programs would probably be considered bad on stylistic grounds (for example, they might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables that are set and never used: these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

5.5 Flow of Control

Lint attempts to detect unreachable portions of program code. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops that can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. Lint also complains about loops which cannot be entered at the top: some valid programs may have such loops, but at best they are bad style and at worst, bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to exit may cause unreachable code which Lint does not detect; the most serious effects of this are in the determination of returned function values, discussed in the next section.

One form of unreachable statement is not usually complained about by Lint: a **break** statement that cannot be reached causes no message. Programs generated by yacc and especially lex may have literally hundreds of unreachable **break** statements. Using the -O switch with the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the Lint output. If these messages are desired, Lint can be invoked with the -b option.

5.6 Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. Lint addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return ( expr );
```

and

```
return ;
```

statements is cause for alarm. In this case, Lint produces the following error message:

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f (a) {
        if (a) return (3);
        g ();
        }
```

Notice that, if a tests false, f will call g and then return with no defined return value; this will trigger a complaint from Lint. If g, like exit, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature. It also accounts for a substantial fraction of the "noise" messages produced by Lint.

On a global scale, Lint detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem.

5.7 Type Checking

Lint enforces the type checking rules of C more strictly than do the compilers. The additional checking is in four major areas:

1. Across certain binary operators and implied assignments

2. At the structure selection operators

3. Between the definition and uses of functions

4. In the use of enumerations

There are a number of operators that have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property. The argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of x's can be intermixed with pointers to x's.

The type checking rules also require that, in structure references, the left operand of a pointer arrow symbol (->) be a pointer to a structure, the left operand of a period '.' be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

5.8 Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where p is a character pointer. Lint quite rightly complains. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his

intentions. It seems harsh for Lint to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint. Otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

5.9 Nonportable Character Use

Lint flags certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
...
if( (c = getchar()) < 0 ) ....
```

works on some machines, but will fail on machines where characters always take on positive values. The real solution is to declare c an integer, since getchar is actually returning integer values. In any case, Lint issues the message:

```
nonportable character comparison
```

A similar issue arises with bitfields. When assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem counter-intuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

5.10 Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use typedefs. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to integer values, losing accuracy. Since there are a number of legitimate reasons for assigning longs to integers, the detection of these assignments is enabled with the **-a** flag.

5.11 Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by Lint. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **-h** flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the star `*' does nothing. This provokes the message "null effect" from Lint. The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange. The test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. In these cases Lint prints the message:

```
degenerate unsigned comparison
```

If one says

```
if( 1 != 0 ) ....
```

Lint reports "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by Lint involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what is intended. The best solution is to parenthesize such expressions, and Lint encourages this

by an appropriate message.

Finally, when the -h flag is in force, Lint complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered bad style, usually unnecessary, and frequently a bug.

5.12 History

There are several forms of older syntax that are discouraged by Lint . These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, ...) could cause ambiguous expressions, such as

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, Lint complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize x to 1. This also caused syntactic difficulties. For example

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { . . .
```

and the compiler must read a fair ways past x in order to sure what the declaration really is. Again, the problem is even more perplexing when the initializer involves a macro.

The current syntax places an equals sign between the variable and the initializer:

```
int  x  =  -1 ;
```

This is free of any possible syntactic ambiguity.

5.13 Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on some machines, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On others, however, double precision values must begin on even word boundaries; thus, not all such assignments make sense. Lint tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the -p or -h flags are in effect.

5.14 Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++] ;
```

will draw the complaint:

```
warning: i evaluation order undefined
```

5.15 Shutting Lint Up

There are occasions when the programmer is smarter than Lint. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by Lint often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with Lint, typically to shut it up, is desirable. Therefore, a number of words are recognized by Lint when they were embedded in comments. Thus, Lint directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the Lint directives don't work.

The first directive is concerned with flow of control information. If a particular place in the program cannot be reached, but this is not apparent to Lint, this can be asserted at the appropriate spot in the program by the directive:

```
/* NOTREACHED */
```

Similarly, if it is desired to turn off strict type checking for the next expression, use the directive:

```
/* NOSTRICT */
```

The situation reverts to the previous default after the next expression. The -v flag can be turned on for one function by the directive:

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by preceding the function definition with the directive:

```
/* VARARGS */
```

In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments that should be

checked. Thus:

```
/* VARARGS2 */
```

causes the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file, discussed in the next section.

5.16 Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined on a library file, but are not used on a source file, draw no complaints. Lint does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, Lint checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the -p flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

5.17 Notes

Lint is by no means perfect. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked; and no attempt is made to match up structure and union declarations across files.

5.18 Current Lint Options

The command currently has the form

```
lint [-options ] files... library-descriptors...
```

The options are

- h Perform heuristic checks
- p Perform portability checks
- v Don't report unused arguments
- u Don't report unused or undefined externals
- b Report unreachable **break** statements.
- x Report unused external declarations
- a Report assignments of **long** to **int** or shorter.
- c Complain about questionable casts
- n No library checking is done
- s Same as **h** (for historical reasons)

CHAPTER 6

ADB: A PROGRAM DEBUGGER

CONTENTS

6.1 Introduction

ADB is an indispensable tool for debugging programs or crashed systems. ADB provides capabilities to look at core files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This chapter provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, and gives examples of printing file system information and of patching.

6.2 Invocation

The ADB invocation syntax is as follows:

adb objectfile corefile

Here objectfile is an executable XENIX file and corefile is a core image file. Often this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are a.out and core, respectively. The filename minus (-) means ignore this argument as in:

```
adb - core
```

ADB has requests for examining locations in either file. A question mark (?) request examines the contents of objectfile; a slash (/) request examines the corefile. The general form of these requests is:

address ? format

or

address / format

6.3 The Current Address - Dot

ADB maintains a pointer to the current address, called dot, similar in function to the current pointer in the editor, ed(1). When an address is entered, the current address is set to that location, so that:

```
0126?i
```

sets dot to octal 126 and prints the instruction at that address. The request

```
.,10/d
```

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the question mark (?) or slash (/) request, the current address can be advanced by typing a newline; it can be decremented by typing a caret (^).

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the following operators:

+	Addition
-	Subtraction
*	Multiplication
%	Integer Division
&	Bitwise AND
|	Bitwise inclusive OR
#	Round up to the next multiple
~	Not

Note that all arithmetic within ADB is 32-bit arithmetic. When typing a symbolic address for a C program, type either "name" or "_name"; ADB recognizes both forms. Because ADB will find only one of "name" and "_name", (generally the first to appear in the source) one will mask the other if they both appear in the same source file.

6.4 Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters:

Letter	Format
b	one byte in octal
c	one byte as a character
o	one word in octal
d	one word in decimal
x	one word in hexadecimal
f	two words in floating point
i	machine instruction
s	a null terminated character string
a	the value of dot
u	one word as unsigned integer
n	print a newline
r	print a blank space
^	backup dot

(Format letters are also available for "long" values, for example, D for long decimal, and F for double floating point.)

6.5 General Request Meanings

The general form of a request is:

```
address,count command modifier
```

which sets "dot" to address and executes the command count times.

The following table illustrates some general ADB command meanings:

Command	Meaning
?	Print contents from a.out file
/	Print contents from core file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or <CONTROL-D>) must be used to exit from ADB.

6.6 Debugging C Programs

The following subsections describe use of ADB in debugging the C programs given in figures at the end of this chapter. Refer to these figures as you work your way through these examples.

6.6.1 Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by charp and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer charp instead of the string pointed to by charp. Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by typing:

```
adb a.out core
```

The first debugging request

```
$c
```

is used to give a C backtrace through the subroutines called. As shown in Figure 2, only one function, main, was called and the arguments argc and argv have hex values 0x2 and 0xlfff90 respectively. Both of these values look reasonable; 0x2 = two arguments, 0xlfff90 = address on stack of parameter vector. These values may be different on your system due to a different mapping of memory.

The next request

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

```
$e
```

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the a.out file is referenced with a question mark (?), whereas the map for the core file is referenced with a slash (/). Furthermore, a good rule of thumb is to use question mark for instructions and slash for data when looking at programs. To print out information about the maps type:

```
$m
```

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by charp. This is done by typing

```
*charp/s
```

which says use charp as a pointer in the core file and print the information as a character string. This printout shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around charp shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function.

```
0x1fff90,3/X
```

prints the hex values of the three consecutive cells pointed to by argv in the function main. Note that these values are the addresses of the arguments to main. Therefore:

```
0x1fffb6/s
```

prints the ASCII value of the first argument. Another way to print this value would have been

```
*"/s
```

The double quote mark (") means ditto, i.e., the the last address typed, in this case 0x1fff90 ; the star (*) instructs ADB to use the address field of the core file as a pointer.

The request

```
.=x
```

prints the current address (not its contents) in hex which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

```
.-10/d
```

6.6.2 Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions "f", "g", and "h" until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

```
adb
```

which assumes the names a.out and core for the executable file and core image file respectively. The request

```
$c
```

fills a page of backtrace references to "f", "g", and "h". Figure 4 shows an abbreviated list (typing <DEL> will terminate the output and bring you back to ADB request level. Additionally, some versions, will automatically quit after 15 levels unless told otherwise with the command:

```
,levelcount$c
```

The request

```
,5$c
```

prints the five most recent activations.

Notice that each function ("f", "g", and "h") has a counter that counts the number of times each has been called.

The request

```
fcnt/D
```

prints the decimal value of the counter for the function f. Similarly "gcnt" and "hcnt" could be printed. Notice that because "fcnt", "gcnt", and "hcnt" are **int** variables, and on the MC68000 **int** is implemented as **long**, to print its value you must use the two word format D.

6.6.3 Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from Software Tools by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6) by typing:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b  [request]
```

The requests

```
settab+8:b
fopen+8:b
tabpos+8:b
```

set breakpoints at the start of these functions. C does not generate statement labels. Therefore, it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as "symbol+8", so that they will appear in any C backtrace since the first two instructions of each function are used to set up the local stack frame. Note that some of the functions are from the C library.

To print the location of breakpoints one types:

```
$b
```

The display indicates a count field. A breakpoint is bypassed count-1 times before causing a stop. The command field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no command fields are present.

By displaying the original instructions at the function settab we see that the breakpoint is set after the tstb instruction, which is the stack probe. We can display the instructions using the ADB request:

```
settab,5?ai
```

This request displays five instructions starting at settab with the addresses of each location displayed. Another variation is

```
settab,5?i
```

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the a.out file with the question (?) command. In general when asking for a printout of multiple items, ADB advances the current address the number of bytes necessary to satisfy the request. In the above example, five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one type:

```
:r
```

To delete a breakpoint, for instance the entry to the function settab, type:

```
settab+8:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for fopen), ADB requests can be used to display the contents of memory. For example:

```
$c
```

to display a stack trace, or:

```
tabs,6/4X
```

to print 6 lines of 4 locations each from the array called tabs. By this time (at location fopen) in the C program, settab has been called and should have set a one in every eighth location of tabs.

The XENIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test

program if

```
:c 0
```

is typed.

6.6.4 Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  ```
  :r arg1 arg2 ... <infile >outfile
  ```

  This request kills any existing program under test and starts the a.out afresh.

- The program being debugged can be single stepped by typing:

  ```
  :s
  ```

  If necessary, this request starts up the program being debugged and stops after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

  ```
  address:r
  ```

- The count field can be used to skip the first n breakpoints as:

  ```
  ,n:r
  ```

  The request

  ```
  ,n:c
  ```

  may also be used for skipping the first n breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by typing:

  ```
  address:c
  ```

- The program being debugged runs as a separate process and can be killed by typing:

```
:k
```

6.7 Maps

XENIX supports several executable file formats. These are used to tell the loader how to load the program file. Nonshared program files are the most common and is generated by a C compiler invocation such as:

```
cc pgm.c
```

A shared file is produced by a C compiler command of the form

```
cc -n pgm.c
```

Note that separate instruction/data files are not supported on the MC68000

ADB interprets these different file formats and provides access to the different segments through a set of maps. To print the maps type:

```
$m
```

In nonshared files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In shared text, the instructions are separated from data and the "?*" accesses the data part of the a.out file. The "?*" request tells ADB to use the second part of the map in the a.out file. Accessing data in the core file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. In shared files the corresponding core file does not contain the program text.

Figure 7 shows the display of three maps for the same program linked as a nonshared and shared respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (0x34 bytes for an a.out file and 02000 bytes for a core file). The "f2" field is the displacement from the beginning of the file to the data. For unshared files with mixed text and data this is the same as the length of the header; for shared files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either a.out or core) is calculated as:

```
b1≤A≤e1 => file address = (A-b1)+f1
b2≤A≤e2 => file address = (A-b2)+f2
```

A user can access locations by using the ADB defined variables. The "$v" request prints the variables initialized by ADB:

b	base address of data segment
d	length of the data segment
s	length of the stack
t	length of the text
m	execution type (407,410,411)

In Figure 7 those variables not present are zero. Use can be made of these variables by expressions such as

```
<b
```

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

```
02000>b
```

that sets b to octal 2000. These variables are useful to know if the file under examination is an executable or core image file.

ADB reads the header of the core image file to find the values for these variables. If the second file specified does not seem to be a core file, or if it is missing then the header of the executable file is used instead.

6.8 Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

6.8.1 Formatted dump

The line

```
<b,-1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down,

the various request pieces mean:

<b The base address of the data segment.

<b,-1 Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format "4o4^8Cn" is broken down as follows:

4o Print 4 octal locations.

4^ Backup the current address 4 locations (to the original start of the field).

8C Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as an at-sign (@) followed by the corresponding character in the range 0140 to 0177. An at-sign is printed as "@@".

n Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, dump, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request

```
120$w
```

sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request

```
4095$s
```

increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The equal sign request (=) can be used to print literal strings. Thus, headings are provided in this dump program with requests of the form:

```
=3n"C Stack Backtrace"
```

This spaces three lines and prints the literal string. The request

```
$v
```

prints all non-zero ADB variables. The request

```
0$s
```

sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 9 shows the results of some formatting requests on the C program of Figure 8.

6.8.2 Directory Dump

As another illustration (Figure 10) consider a set of requests to dump the contents of a directory (which is made up of an integer inumber followed by a 14 character name):

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example, the **u** prints the inumber as an unsigned decimal integer, the "8t" means that ADB will space to the next multiple of 8 on the output line, and the "14c" prints the 14 character file name.

6.8.3 Ilist Dump

Similarly the contents of the ilist of a file system, (e.g., /dev/src) could be dumped with the following set of requests:

```
adb /dev/src -
02000>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3bn",...
```

(Note that the two lines separated by ellipses should be entered as one line with no intervening space. The line is broken here so that it will fit on the page.) In this example the value of the base for the map was changed to 02000 by typing

```
?m<b
```

since that is the start of an ilist within a file system. "Brd" above was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the "2Y" operator. Figure 10 shows portions of these requests as applied to a directory and file system.

6.8.4 Converting values

ADB may be used to convert values from one representation to another. For example:

```
072 = odx
```

prints

```
072     58      #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

prints

```
a       0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6.9 Patching

Patching files with ADB is accomplished with the write (w or W) request. This is often used in conjunction with the locate, (l or L) request. In general, the request syntax for l and w are similar:

```
?l value
```

The request l is used to match on two bytes; L is used for four bytes. The request w is used to write two bytes, whereas W writes four bytes. The value field in either locate or write requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, file1 and file2 are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 8. We can change the word "This" to "The " in the executable file for this program, ex7, by using the following requests:

```
adb -w ex7 -
?1 'Th'
?W 'The'
```

The request

```
?1
```

starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of the question mark (?) to write to the a.out file. The form "?*" would have been used for a 411 file.

More frequently the request will be typed as:

```
?1 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The ":s" request is normally used to single step through a process or start a process in single step mode. In this case it starts a.out as a subprocess with arguments arg1 and arg2. If there is a subprocess running ADB writes to it rather than to the file so the w request causes flag to be changed in the memory of the subprocess.

6.10 Notes

Below is a list of some things that users should be aware of:

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.

2. When printing addresses, ADB uses either text or data symbols from the a.out file. This sometimes causes

unexpected symbol names to be printed with data (e.g., "savr5+022"). This does not happen if question mark (?) is used for text (instructions) and slash (/) for data.

3. Local variables cannot be addressed.

6.11 Figures

Figure 1: C program with pointer bug

```
#include <stdio.h>
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp = "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
        char    cc;
        FILE *file;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((file = fopen(argv[1],"w")) == NULL){
                printf("%s : can't open\n", argv[1]);
                exit(8);
        }
        charp = 'T';
printf("debug 1 %s\n",charp);
        while(cc=  *charp++)
                putc(cc,file);
        fflush(file);
}
```

Figure 2: ADB output for C program of figure 1

```
adb
$c
start+44:       _main   (0x2, 0x1FFF90)
$r
d0      0x0                         a0      0x54
d1      0x8                         a1      0x1FFF90
d2      0x0                         a2      0x0
d3      0x0                         a3      0x0
d4      0x0                         a4      0x0
d5      0x0                         a5      0x0
d6      0x0                         a6      0x1FFF7C
d7      0x0                         sp      0x1FFF74

ps      0x0
pc      0x80E4  _main+160:          movb    (a0),-1.(a6)
$e
_environ:       0x1FFF9C
_errno: 0x19
_bb:    0x0
_obuf:  0x0
_charp: 0x55
__iob:  0x9B1C
__sobuf:        0x64656275
__lastbu:       0x96F8
__sibuf:        0x0
_allocs:        0x0
_allocp:        0x0
_alloct:        0x0
_allocx:        0x0
_end:   0x0
_edata: 0x0
$m
? map   `a.out'
b1 = 0x8000     e1 = 0x970C     f1 = 0x20
b2 = 0x8000     e2 = 0x970C     f2 = 0x20
/ map   `-'
b1 = 0x0        e1 = 0x1000000  f1 = 0x0
b2 = 0x0        e2 = 0x0        f2 = 0x0
*charp/s
0x55:
data address not found
0x1fff90,3/X
0x1FFF90:       0x1FFFB0        0x1FFFB6        0x0
0x1fffb0/s
0x1FFFB0:       a.out
/s
0x1FFFB0:       a.out
.=X
                0x1FFFB0
```

```
.-10/d
0x1FFFA6:       65497
$q
```

Figure 3: Multiple function C program

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}


g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}


f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}


main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
$c
_h+46:          _f      (0x2, 0x92D)
_g+48:          _h      (0x92C, 0x92B)
_f+70:          _g      (0x92D, 0x1258)
_h+46:          _f      (0x2, 0x92B)
_g+48:          _h      (0x92A, 0x929)
_f+70:          _g      (0x92B, 0x1254)
_h+46:          _f      (0x2, 0x929)
_g+48:          _h      (0x928, 0x927)
<INTERRUPT>
adb
,5$c
_h+46:          _f      (0x2, 0x92D)
_g+48:          _h      (0x92C, 0x92B)
_f+70:          _g      (0x92D, 0x1258)
_h+46:          _f      (0x2, 0x92B)
_g+48:          _h      (0x92A, 0x929)
fcnt/D
_fcnt:          1175
gcnt/D
_gcnt:          1174
hcnt/D
_hcnt:          1174
$q
```

Figure 5: C program to decode tabs

```
#include <stdio.h>
#define MAXLINE 80
#define YES             1
#define NO              0
#define TABSP           8
char    input[] = "data";
char    ibuf[518];
int     tabs[MAXLINE];



main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);   /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getch(ibuf)) != -1) {
                switch(c) {
                        case '\t':      /* TAB */
                                while(tabpos(col) != YES) {
                                        /* put BLANK */
                                        putchar(' ');
                                        col++ ;
                                }
                                break;
                        case '\n':      /*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}
```

```
/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}



/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}



/* getch(ibuf) - Just do a getc call, but not a macro */
getch(ibuf)
FILE *ibuf;
{
        return(getc(ibuf));
}
```

Figure 6: ADB output for C program of Figure 5

```
adb a.out
settab+8:b
fopen+8:b
getch+8:b
tabpos+8:b
$b
breakpoints
count    bkpt                command
1        _tabpos+8
1        _getch+8
1        _fopen+8
1        _settab+8
settab,5?ia
_settab:                     link    a6,#0xFFFFFFFC
_settab+4:                   tstb    -132.(a7)
_settab+8:                   moveml  #<>,-(a7)
_settab+12:                  clrl    -4.(a6)
_settab+16:                  cmpl    #0x50,-4.(a6)
_settab+24:
settab,5?i
_settab:                     link    a6,#0xFFFFFFFC
                             tstb    -132.(a7)
                             moveml  #<>,-(a7)
                             clrl    -4.(a6)
                             cmpl    #0x50,-4.(a6)
:r
a.out:running
breakpoint      _settab+8:       moveml  #<>,-(a7)
settab+8:d
:c
a.out:running
breakpoint      _fopen+8:        jsr     __findio
$c
_main+52:       _fopen  (0x9750, 0x9958)
start+44:       _main   (0x1, 0x1FFF98)
tabs,6/4X
_tabs:  0x1     0x0      0x0      0x0
        0x0     0x0      0x0      0x0
        0x1     0x0      0x0      0x0
        0x0     0x0      0x0      0x0
        0x1     0x0      0x0      0x0
        0x0     0x0      0x0      0x0
```

Figure 7: ADB output for maps

```
adb a.out.unshared core.unshared
$m
? map    `a.out.unshared'
b1 = 0x8000      e1 = 0x83E4      f1 = 0x20
b2 = 0x8000      e2 = 0x83E4      f2 = 0x20
/ map    `core.unshared'
b1 = 0x8000      e1 = 0x8800      f1 = 0x800
b2 = 0x1EB000    e2 = 0x200000    f2 = 0x1000
$v
variables
b = 0x8000
d = 0x800
e = 0x8000
m = 0x107
s = 0x15000
$q


adb a.out.shared core.shared
$m
? map    `a.out.shared'
b1 = 0x8000      e1 = 0x8390      f1 = 0x20
b2 = 0x10000     e2 = 0x10054     f2 = 0x3B0
/ map    `core.shared'
b1 = 0x10000     e1 = 0x10108     f1 = 0x800
b2 = 0x1EB000    e2 = 0x200000    f2 = 0x1000
$v
variables
b = 0x10390
d = 0x800
e = 0x8000
m = 0x108
s = 0x15000
$q
```

Figure 8: Simple C program illustrating formatting and patching

```
char    str1[]  = "This is a character string";
int     one     = 1;
int     number  = 456;
long    lnum    = 1234;
float   fpt     = 1.25;
char    str2[]  = "This is the second character string";
main()
{
        one = 2;
}
```

Figure 9: ADB output illustrating fancy formats

```
adb a.out.shared core.shared
<b,-1/8ona
_str1:      052150 064563 020151 071440 060440 061550 060562 060543

_str1+16:   072145 071040 071564 071151 067147 0      0      01

_number:
_number:    0      0710   0      02322  037640 0      052150 064563

_str2+4:    020151 071440 072150 062440 071545 061557 067144 020143

_str2+20:   064141 071141 061564 062562 020163 072162 064556 063400

$nd:
$nd:        01     0140
<b,20/4o4^8Cn
_str1:      052150 064563 020151 071440  This is
            060440 061550 060562 060543  a charac
            072145 071040 071564 071151  ter stri
            067147 0      0      01      ng@`@`@`@`@`@a

_number:    0     0710    0      02322   @`@`@aH@`@`@dR

_fpt:       037640 0      052150 064563  ? @`@`This
            020151 071440 072150 062440  is the
            071545 061557 067144 020143  second c
            064141 071141 061564 062562  haracter
            020163 072162 064556 063400   string@`

$nd:        01     0140
data address not found
<b,20/4o4^8t8Cna
_str1:      052150 064563 020151 071440          This is
_str1+8:    060440 061550 060562 060543          a charac
_str1+16:   072145 071040 071564 071151          ter stri
_str1+24:   067147 0      0      01              ng@`@`@`@`@`@a
_number:
_number:    0      0710   0      02322           @`@`@aH@`@`@dR
_fpt:
_fpt:       037640 0      052150 064563          ? @`@`This
_str2+4:    020151 071440 072150 062440           is the
_str2+12:   071545 061557 067144 020143          second c
_str2+20:   064141 071141 061564 062562          haracter
_str2+28:   020163 072162 064556 063400           string@`
$nd:
$nd:        01     0140
data address not found
<b,10/2b8t^2cn
_str1:      0124   0150          Th
```

```
0151    0163        is
040     0151        i
0163    040         s
0141    040         a
0143    0150        ch
0141    0162        ar
0141    0143        ac
0164    0145        te
0162    040         r
$q
```

Figure 10: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"
0,-1?utl4cn

                Inode   Name
0:              652     .
                82      ..
                5971    cap.c
                5323    cap
                0       pp


adb /dev/src -
02000>b
?m<b
new map     `/dev/src'
b1 = 02000          e1 = 0100000000     f1 = 0
b2 = 0              e2 = 0              f2 = 0
$v
variables
b = 02000
<b,-1?"flags"8ton"links,uid,gid"8t3bn"
size"8tbrdn"addr"8t8un"times"8t2Y2na
(type above two lines all on one line)
02000:  flags 073145
        links,uid,gid 0163  0164    0141
        size  0162    10356
        addr  28770   8236  25956 27766 25455 8236  25956 25206
        times 1976 Feb 5 08:34:56     1975 Dec 28 10:55:15

02040:  flags 024555
        links,uid,gid 012   0163  0164
        size  0162    25461
        addr  8308  * 30050 8294  25130 15216 26890 29806 10784
        times 1976 Aug 17 12:16:51    1976 Aug 17 12:16:51

02100:  flags 05173
        links,uid,gid 011   0162  0145
        size  0147    29545
        addr  25972   8306  28265 8308  25642 15216 2314  25970
        times 1977 Apr 2 08:58:01     1977 Feb 5 10:21:44
```

6.12 ADB Summary

Command Summary

a. Formatted printing

? format	print from a.out file according to format
/ format	print from core file according to format
= format	print the value of dot
?w expr	write expression into a.out file
/w expr	write expression into core file
?l expr	locate expression in a.out file

b. Breakpoint and program control

:b	set breakpoint at dot
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run a.out file under ADB control
:s	single step

c. Miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d. Calling the shell

!	call shell to read rest of line

e. Assignment to variables

>name	assign dot to variable or register name

6.12.1 Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	machine instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next n space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

6.12.2 Expression Summary

a. Expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b. Dyadic operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c. Monadic operators

~	not
*	contents of location
-	integer negation