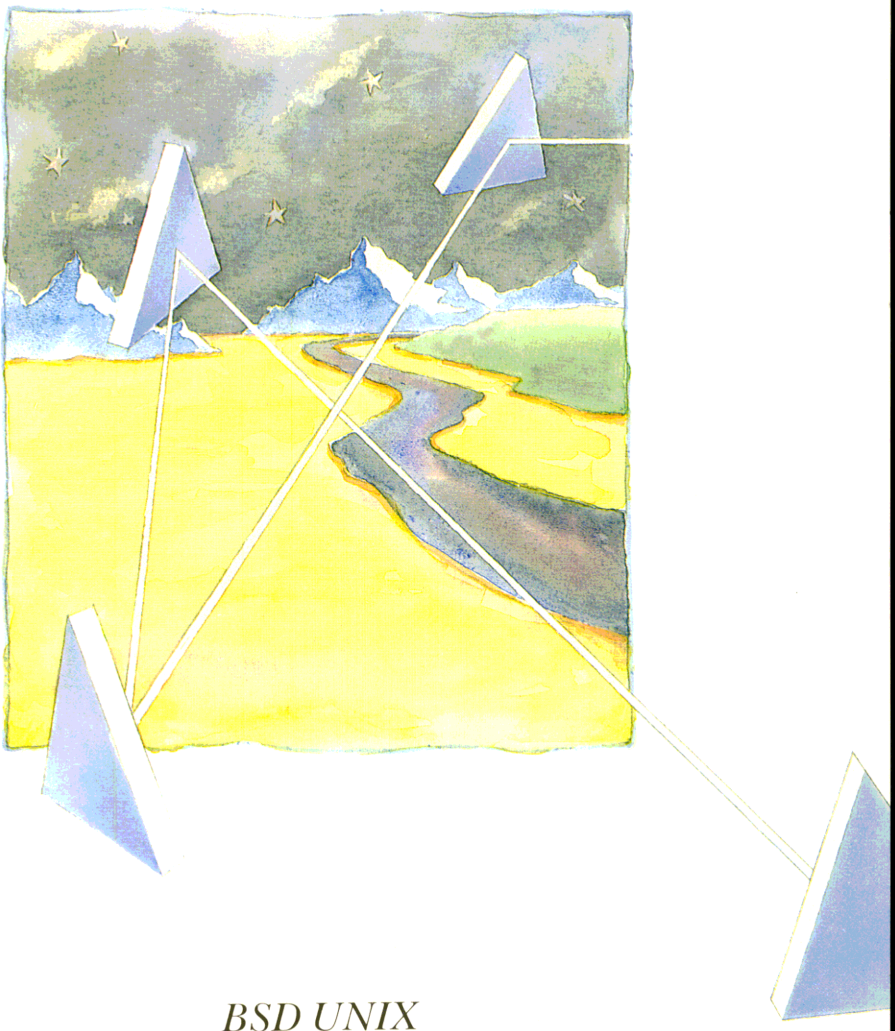


BSD UNIX
Programmer's Manual

apollo

BSD UNIX Programmer's Manual

Order No. 017272-A00

First Printing: November 1989

This manual reflects system enhancements made at Berkeley and sponsored in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in these documents are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

10 9 8 7 6 5 4 3 2 1

Preface

The *BSD UNIX Programmer's Manual* describes programming utilities available in Domain/OS BSD UNIX. This manual is intended for programmers who are familiar with BSD UNIX software and Domain/OS. It provides neither a general overview of Domain/OS BSD nor details of the implementation of the system. We assume that you are already familiar with the material in *Using Your BSD Environment*.

We've organized this manual as follows:

Chapter 1	Provides an IPC (interprocess communication) tutorial.
Chapter 2	Describes advanced BSD IPC.
Chapter 3	Describes the **dbx** source code debugger.
Chapter 4	Describes the **rcs** version control system.
Chapter 5	Describes **sccs** (source code control system).
Chapter 6	Describes the **yacc** compiler.
Chapter 7	Describes the **lex** lexical analysis program generator.
Chapter 8	Describes the **m4** macro processor.
Chapter 9	Describes the **curses** utility.
Appendix A	Describes the **make** utility.
Appendix B	Describes the **mk** utility.

Related Manuals

The file **/install/doc/apollo/os.v.***latest software release number***__manuals** lists current titles and revisions for all available manuals.

For example, at Software Release 10.2 (SR10.2) refer to the file **/install/doc/apollo/os.v.10.2__manuals** to check that you are using the correct version of manuals. You may also want to use this file to check that you have ordered all of the manuals that you need.

The same information is available online. In the UNIX environment, type **man manuals**. In the Aegis environment, type **help manuals**.

Refer to the *Apollo Documentation Quick Reference* (002685) and the *Domain Documentation Master Index* (011242) for a complete list of related documents.

For introductory information about the Domain/OS system and details about using the BSD environment, refer to the following documents:

- *Getting Started with Domain/OS* (2348)
- *Using Your BSD Environment* (11020)
- *BSD UNIX User's Manual*, Volumes I and II (017271 and 017623)
- *BSD Command Reference* (005800)
- *Domain Display Manager Command Reference* (11418)

For more information on programming in the Domain/OS BSD environment, refer to the following documents:

- *Domain/OS Call Reference*, Volumes 1 and 2 (7196 and 12888)
- *Domain/OS Programming Environment Reference* (11010)
- *Domain Binder and Librarian Reference* (4977)
- *Domain C Language Reference* (2093)
- *BSD Programmer's Manual* (017272)
- *BSD Programmer's Reference* (005801)
- *Managing BSD System Software* (10853)

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo Product Reporting (APR) system for comments related to hardware, software, and documentation. By using this formal channel you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis, BSD, or SysV). Refer to the **mkapr** shell command description. You can view the same description online by typing:

$ man 1 mkapr (in the SysV environment)

% man 1 mkapr (in the BSD environment)

$ help mkapr (in the Aegis environment)

Alternatively, you may use the Reader's Response form at the back of this manual to submit comments about the manual.

BSD UNIX Programmer's Manual

4.3 Berkeley Software Distribution

This volume contains documents which supplement the manual pages in the *BSD Programmer's Reference*.

General Reference

Programming Tools

Programming Libraries

Appendices

An Introductory 4.3BSD Interprocess Communication Tutorial

Stuart Sechrest

Computer Science Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

ABSTRACT

Berkeley UNIX† 4.3BSD offers several choices for interprocess communication. To aid the programmer in developing programs which are comprised of cooperating processes, the different choices are discussed and a series of example programs are presented. These programs demonstrate in a simple way the use of pipes, socketpairs, sockets and the use of datagram and stream communication. The intent of this document is to present a few simple example programs, not to describe the networking system in full.

1. Goals

Facilities for interprocess communication (IPC) and networking were a major addition to UNIX in the Berkeley UNIX 4.2BSD release. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In UNIX a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing, and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the 'ls' command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in UNIX for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

The use of descriptors is not the only communication interface provided by UNIX. The signal mechanism sends a tiny amount of information from one process to another. The signaled process receives only the signal type, not the identity of the sender, and the number of possible signals is small. The signal semantics limit the flexibility of the signaling mechanism as a means of interprocess communication.

The identification of IPC with I/O is quite longstanding in UNIX and has proved quite successful. At first, however, IPC was limited to processes communicating within a single machine. With Berkeley UNIX 4.2BSD this expanded to include IPC between machines. This expansion has necessitated some change in the way that descriptors are created. Additionally, new possibilities for the meaning of read and write have been admitted. Originally the meanings, or semantics, of these terms were fairly simple. When you wrote something it was delivered. When you read something, you were blocked until the data arrived. Other possibilities exist, however. One can write without full assurance of delivery if one can check later to catch occasional failures. Messages can be kept as discrete units or merged into a stream. One can ask to read, but insist on not waiting if nothing is immediately available. These new possibilities are allowed in the Berkeley UNIX IPC interface.

---

† UNIX is a trademark of AT&T Bell Laboratories.

Thus Berkeley UNIX 4.3BSD offers several choices for IPC. This paper presents simple examples that illustrate some of the choices. The reader is presumed to be familiar with the C programming language [Kernighan & Ritchie 1978], but not necessarily with the system calls of the UNIX system or with processes and interprocess communication. The paper reviews the notion of a process and the types of communication that are supported by Berkeley UNIX 4.3BSD. A series of examples are presented that create processes that communicate with one another. The programs show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work. They can, therefore, serve as models for the programmer trying to construct programs which are comprised of cooperating processes.

2. Processes

A *program* is both a sequence of statements and a rough way of referring to the computation that occurs when the compiled statements are run. A *process* can be thought of as a single line of control in a program. Most programs execute some statements, go through a few loops, branch in various directions and then end. These are single process programs. Programs can also have a point where control splits into two independent lines, an action called *forking*. In UNIX these lines can never join again. A call to the system routine *fork()*, causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value returned by *fork()*. The parent will receive a process id for the child, the child will receive a zero. Calls to *fork()*, therefore, typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets (sockets are communication objects that will be discussed below). Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, *stdin*, *stdout* and *stderr*. These are the standard input, output and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal (devices are also treated as files in UNIX), the child's input will be taken from the same terminal. Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

3. Pipes

Most users of UNIX know that they can pipe the output of a program "prog1" to the input of another, "prog2," by typing the command *"prog1 | prog2."* This is called "piping" the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example, *"prog1,"* the shell forks a process, which executes the program, prog1, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command, *"prog1 | prog2,"* the shell creates two processes connected by a pipe. One process runs the program, prog1, the other runs prog2. The pipe is an I/O mechanism with two ends, or sockets. Data that is written into one socket can be read from the other.

Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at *stdout* and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite end of the pipe for *stdin*.

Let us now examine a program that creates a pipe for communication between its child and itself (Figure 1). A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

In Figure 1, the parent process makes a call to the system routine *pipe()*. This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. *Pipe()* is passed an array into which it places the index numbers of the sockets it created. The two ends are not

```
#include <stdio.h>

#define DATA "Bright star, would I were steadfast as thou art . . ."

/*
 * This program creates a pipe, then forks. The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device. I can write to the output socket (sockets[1], the second socket
 * of the array returned by pipe()) and read from the input socket
 * (sockets[0]), but not vice versa.
 */

main()
{
        int sockets[2], child;

        /* Create a pipe */
        if (pipe(sockets) < 0) {
                perror("opening stream socket pair");
                exit(10);
        }

        if ((child = fork()) == -1)
                perror("fork");
        else if (child) {
                char buf[1024];

                /* This is still the parent. It reads the child's message. */
                close(sockets[1]);
                if (read(sockets[0], buf, 1024) < 0)
                        perror("reading message");
                printf("-->%s\n", buf);
                close(sockets[0]);
        } else {
                /* This is the child. It writes a message to its parent. */
                close(sockets[0]);
                if (write(sockets[1], DATA, sizeof(DATA)) < 0)
                        perror("writing message");
                close(sockets[1]);
        }
}
```

Figure 1 Use of a pipe

equivalent. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling *fork()*. Figure 2 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

Just what is a pipe? It is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes,

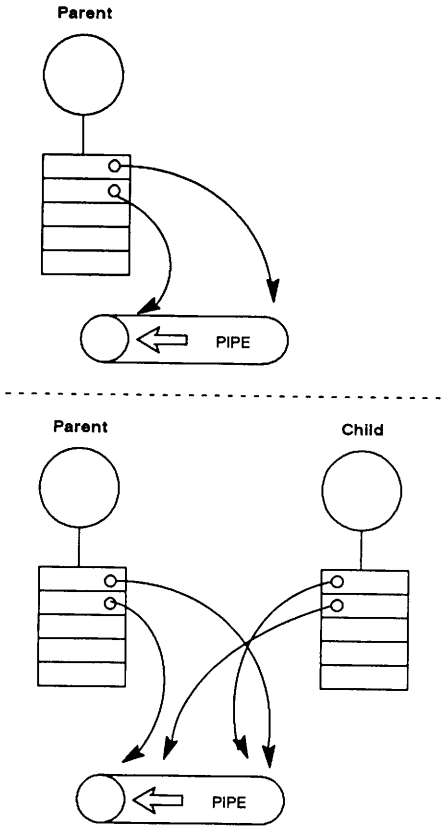


Figure 2 Sharing a pipe between parent and child

one for use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. It is not required that unused descriptors be closed, but it is good practice.) A pipe is also a *stream* communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, he is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to *write()* or from several calls to *write()* which were concatenated.

4. Socketpairs

Berkeley UNIX 4.3BSD provides a slight generalization of pipes. A pipe is a pair of connected sockets for one-way stream communication. One may obtain a pair of connected sockets for two-way stream communication by calling the routine *socketpair()*. The program in Figure 3 calls *socketpair()* to create such a connection. The program uses the link for communication in both directions. Since socketpairs are an extension of pipes, their use resembles that of pipes. Figure 4 illustrates the result of a fork

following a call to *socketpair()*.

Socketpair() takes as arguments a specification of a domain, a style of communication, and a protocol. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain is a space of names that may be bound to sockets and implies certain other conventions. Currently, socketpairs have only been implemented for one domain, called the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows communication between sockets on the same machine.

Note that the header files *<sys/socket.h>* and *<sys/types.h>*. are required in this program. The constants AF_UNIX and SOCK_STREAM are defined in *<sys/socket.h>*, which in turn requires the file *<sys/types.h>* for some of its definitions.

5. Domains and Protocols

Pipes and socketpairs are a simple solution for communicating between a parent and child or between child processes. What if we wanted to have processes that have no common ancestor with whom to set up communication? Neither standard UNIX pipes nor socketpairs are the answer here, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines. In Berkeley UNIX 4.3BSD one can create individual sockets, give them names and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain*. There are several domains for sockets. Two that will be used in the examples here are the UNIX domain (or AF_UNIX, for Address Format UNIX) and the Internet domain (or AF_INET). UNIX domain IPC is an experimental facility in 4.2BSD and 4.3BSD. In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to the socket by giving the proper pathname. UNIX domain names, therefore, allow communication between any two processes that work in the same file system. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines.

Communication follows some particular "style." Currently, communication is either through a *stream* or by *datagram*. Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *write()* or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections and transfers data between sockets, perhaps sending the data

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."

/*
 * This program creates a pair of connected sockets then forks and
 * communicates over them.  This is very similar to communication with pipes,
 * however, socketpairs are two-way communications objects. Therefore I can
 * send messages in both directions.
 */

main()
{
      int sockets[2], child;
      char buf[1024];

      if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
            perror("opening stream socket pair");
            exit(1);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) { /* This is the parent. */
            close(sockets[0]);
            if (read(sockets[1], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
                  perror("writing stream message");
            close(sockets[1]);
      } else {          /* This is the child. */
            close(sockets[1]);
            if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
                  perror("writing stream message");
            if (read(sockets[0], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      }
}
```

Figure 3 Use of a socketpair

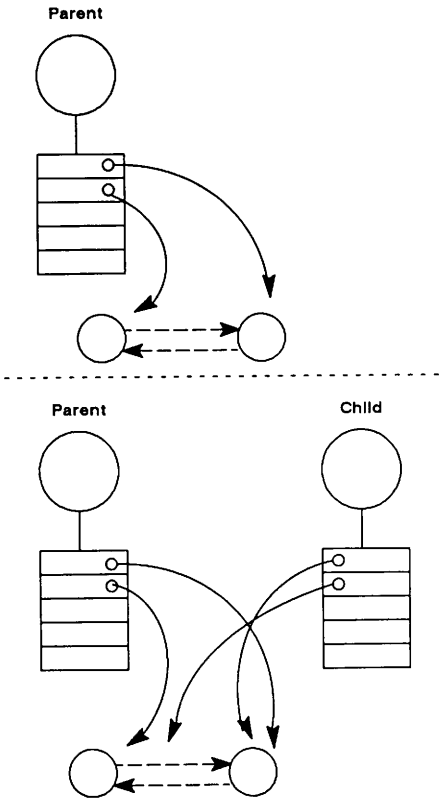


Figure 4 Sharing a socketpair between parent and child

across a network. This code also keeps track of the names that are bound to sockets. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

One specifies the domain, style and protocol of a socket when it is created. For example, in Figure 5a the call to *socket()* causes the creation of a datagram socket with the default protocol in the UNIX domain.

6. Datagrams in the UNIX Domain

Let us now look at two programs that create sockets separately. The programs in Figures 5a and 5b use datagram communication rather than a stream. The structure used to name UNIX domain sockets is defined in the file *<sys/un.h>*. The definition has also been included in the example for clarity.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*
 * In the included file <sys/un.h> a sockaddr_un is defined as follows
 * struct sockaddr_un {
 *     short sun_family;
 *     char  sun_path[108];
 * };
 */

#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket, binds a name to it,
 * then reads from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_un name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, NAME);
      if (bind(sock, &name, sizeof(struct sockaddr_un))) {
            perror("binding name to datagram socket");
            exit(1);
      }
      printf("socket -->%s\n", NAME);
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
      unlink(NAME);
}
```

Figure 5a Reading UNIX domain datagrams

Each program creates a socket with a call to *socket()*. These sockets are in the UNIX domain. Once a name has been decided upon it is attached to a socket by the system call *bind()*. The program in Figure 5a uses the name "socket", which it binds to its socket. This name will appear in the working directory of the program. The routines in Figure 5b use its socket only for sending messages. It does not create a name for the socket because no other process has to refer to it.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is udgramsend pathname
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un name;

      /* Create socket on which to send. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Construct name of socket to send to. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, argv[1]);
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0,
          &name, sizeof(struct sockaddr_un)) < 0) {
            perror("sending datagram message");
      }
      close(sock);
}
```

Figure 5b Sending a UNIX domain datagrams

Names in the UNIX domain are path names. Like file path names they may be either absolute (e.g. "/dev/imaginary") or relative (e.g. "socket"). Because these names are used to allow processes to rendezvous, relative path names can pose difficulties and should be used with care. When a name is bound into the name space, a file (inode) is allocated in the file system. If the inode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause directories to fill up with these objects. The names are removed by calling *unlink()* or using the *rm*(1) command. Names in the UNIX domain are only used for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

There is no established means of communicating names to interested parties. In the example, the program in Figure 5b gets the name of the socket to which it will send its message through its command line arguments. Once a line of communication has been created, one can send the names of additional, perhaps new, sockets over the link. Facilities will have to be built that will make the distribution of names less of a problem than it now is.

7. Datagrams in the Internet Domain

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * In the included file <netinet/in.h> a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *     short sin_family;
 *     u_short     sin_port;
 *     struct in_addr sin_addr;
 *     char  sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_in name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 0;
      if (bind(sock, &name, sizeof(name))) {
            perror("binding datagram socket");
            exit(1);
      }
      /* Find assigned port value and print it out. */
      length = sizeof(name);
      if (getsockname(sock, &name, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(name.sin_port));
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
}
```

Figure 6a Reading Internet domain datagrams

The examples in Figure 6a and 6b are very close to the previous example except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the file *<netinet/in.h>*. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple *<protocol, local machine address, local port, remote machine address, remote port>*. An association may be transient when using datagram sockets; the association actually exists during a *send* operation.

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY. The wildcard value is used in the program in Figure 6a. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from "anywhere," but one cannot send a message "anywhere." The program in Figure 6b is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one's messages. Certain daemons, offering certain advertised services, have reserved or "well-known" port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit *bind* call is made with a port number of 0, or when a *connect* or *send* is performed on an unbound socket. Note that port numbers are not automatically reported back to the user. After calling *bind()*, asking for port 0, one may call *getsockname()* to discover what port was actually assigned. The routine *getsockname()* will not work for names in the UNIX domain.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by *getsockname()* may result in a misinterpretation. To print out the number, it is necessary to use the routine *ntohs()* (for *network to host: short*) to convert the number from the network representation to the host's representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called *htons()*; similar routines exist for long integers. For further information, refer to the entry for *byteorder* in section 3 of the manual.

8. Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. Figures 7a and 7b show two programs that create such a connection. The program in 7a is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls *connect()*, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a SIGPIPE signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see the manual page for *signal* or *sigvec*), the process will terminate and the shell will print the message "broken pipe."

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is dgramsend hostname
 * portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in name;
      struct hostent *hp, *gethostbyname();

      /* Create socket on which to send. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /*
       * Construct name, with no wildcards, of the socket to send to.
       * Getnostbyname() returns a structure including the network address
       * of the specified host.  The port number is taken from the command
       * line.
       */
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
      name.sin_family = AF_INET;
      name.sin_port = htons(atoi(argv[2]));
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
            perror("sending datagram message");
      close(sock);
}
```

Figure 6b Sending an Internet domain datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line.  One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is streamwrite hostname portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp, *gethostbyname();
      char buf[1024];

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sin_family = AF_INET;
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
      server.sin_port = htons(atoi(argv[2]));

      if (connect(sock, &server, sizeof(server)) < 0) {
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
      close(sock);
}
```

Figure 7a Initiating an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
      int sock, length;
      struct sockaddr_in server;
      int msgsock;
      char buf[1024];
      int rval;
      int i;

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using wildcards */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 0;
      if (bind(sock, &server, sizeof(server))) {
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(server.sin_port));

      /* Start accepting connections */
      listen(sock, 5);
      do {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
                  if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                  i = 0;
                  if (rval == 0)
```

```
                        printf("Ending connection\n");
                else
                        printf("-->%s\n", buf);
            } while (rval != 0);
            close(msgsock);
        } while (TRUE);
        /*
         * Since this program has an infinite loop, the socket "sock" is
         * never explicitly closed.  However, all sockets will be closed
         * automatically when a process is killed or terminates normally.
         */
}
```

Figure 7b Accepting an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select() to check that someone is trying to connect
 * before calling accept().
 */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;
        fd_set ready;
        struct timeval to;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
                perror("binding stream socket");
                exit(1);
        }
        /* Find out assigned port number and print it out */
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
                perror("getting socket name");
                exit(1);
```

```
        }
        printf("Socket has port #%d\n", ntohs(server.sin_port));

        /* Start accepting connections */
        listen(sock, 5);
        do {
              FD_ZERO(&ready);
              FD_SET(sock, &ready);
              to.tv_sec = 5;
              if (select(sock + 1, &ready, 0, 0, &to) < 0) {
                    perror("select");
                    continue;
              }
              if (FD_ISSET(sock, &ready)) {
                    msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
                    if (msgsock == -1)
                          perror("accept");
                    else do {
                          bzero(buf, sizeof(buf));
                          if ((rval = read(msgsock, buf, 1024)) < 0)
                                perror("reading stream message");
                          else if (rval == 0)
                                printf("Ending connection\n");
                          else
                                printf("-->%s\n", buf);
                    } while (rval > 0);
                    close(msgsock);
              } else
                    printf("Do something else\n");
        } while (TRUE);
}
```

Figure 7c Using select() to check for pending connections

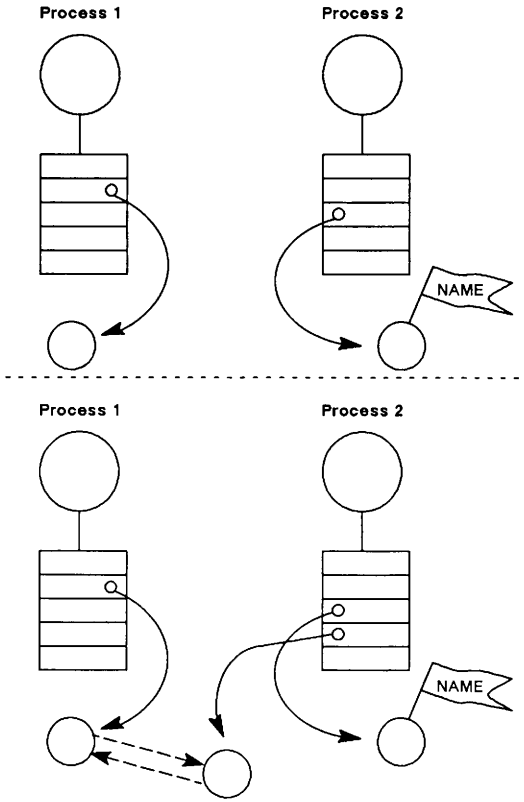


Figure 8 Establishing a stream connection

Forming a connection is asymmetrical; one process, such as the program in Figure 7a, requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 8. Process 2 has created a socket and bound a port number to it. Process 1 has created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

The program in Figure 7b is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls *listen()* for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. *Listen()* marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If

the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of *listen()*; the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 8 shows the result of Process 1 connecting with the named socket of Process 2, and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The *accept()* call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

The program in Figure 7c is a slight variation on the server in Figure 7b. It avoids blocking when there are no pending connection requests by calling *select()* to check for pending requests before calling *accept()*. This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

The programs in Figures 9a and 9b show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single file system. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this paper.

9. Reads, Writes, Recvs, etc.

UNIX 4.3BSD has several system calls for reading and writing information. The simplest calls are *read()* and *write()*. *Write()* takes as arguments the index of a descriptor, a pointer to a buffer containing the data and the size of the data. The descriptor may indicate either a file or a connected socket. "Connected" can mean either a connected stream socket (as described in Section 8) or a datagram socket for which a *connect()* call has provided a default destination (see the *connect()* manual page). *Read()* also takes a descriptor that indicates either a file or a socket. *Write()* requires a connected socket since no destination is specified in the parameters of the system call. *Read()* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *read()* and *write()* that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are *readv()* and *writev()*, for read and write *vector*.

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a SIGURG signal, if this signal has been enabled (see the manual page for *signal* or *sigvec*). See [Leffler 1986] for a more complete description of the OOB mechanism. There are a pair of calls similar to *read* and *write* that allow options, including sending and receiving OOB information; these are *send()* and *recv()*. These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. These calls also allow *peeking* at data in a stream. That is, they allow a process to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as *read()* and *write()*.

To send datagrams, one must be allowed to specify the destination. The call *sendto()* takes a destination address as an argument and is therefore used for sending datagrams. The call *recvfrom()* is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use *read()* or *recv()*.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the command line and sends a
 * one line message to that socket. The form of the command line is
 * ustreamwrite pathname
 */
main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, argv[1]);

      if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
            close(sock);
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
}
```

Figure 9a Initiating a UNIX domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket's name it begins a loop. Each time through the
 * loop it accepts a connection and prints out messages from it.  When the
 * connection breaks, or a termination message comes through, the program
 * accepts a new connection.
 */
```

```
main()
{
      int sock, msgsock, rval;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using file system name */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, NAME);
      if (bind(sock, &server, sizeof(struct sockaddr_un))) {
            perror("binding stream socket");
            exit(1);
      }
      printf("Socket has name %s\n", server.sun_path);
      /* Start accepting connections */
      listen(sock, 5);
      for (;;) {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
                  if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                  else if (rval == 0)
                        printf("Ending connection\n");
                  else
                        printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
      }
      /*
       * The following statements are not executed, because they follow an
       * infinite loop.  However, most ordinary programs will not run
       * forever.  In the UNIX domain it is necessary to tell the file
       * system that one is through using NAME.  In most programs one uses
       * the call unlink() as below. Since the user will have to kill this
       * program, it will be necessary to remove the name by a command from
       * the shell.
       */
      close(sock);
      unlink(NAME);
}
```

Figure 9b Accepting a UNIX domain stream connection

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are *sendmsg()* and *recvmsg()*. These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

The various options for reading and writing are shown in Figure 10, together with their parameters. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this paper, the calls *read()* and *write()* have been used whenever possible.

10. Choices

This paper has presented examples of some of the forms of communication supported by Berkeley UNIX 4.3BSD. These have been presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

```
/*
 * The variable descriptor may be the descriptor of either a file
 * or of a socket.
 */
cc = read(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

/*
 * An iovec can include several source buffers.
 */
cc = readv(descriptor, iov, iovcnt)
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = writev(descriptor, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

/*
 * The variable ''sock'' must be the descriptor of a socket.
 * Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Figure 10 Varieties of read and write commands

Pipes have the advantage of portability, in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common process. They do not allow intermachine communication.

The two communication domains, UNIX and Internet, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagrams and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a process is only allowed a limited number of open streams, as there are usually only 64 entries available in the open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often be the deciding factor in favor of streams.

11. What to do Next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

An introduction to the UNIX system and programming using UNIX system calls can be found in [Kernighan and Pike 1984]. Further documentation of the Berkeley UNIX 4.3BSD IPC mechanisms can be found in [Leffler et al. 1986]. More detailed information about particular calls and protocols is provided in sections 2, 3 and 4 of the UNIX Programmer's Manual [CSRG 1986]. In particular the following manual pages are relevant:

creating and naming sockets	socket(2), bind(2)
establishing connections	listen(2), accept(2), connect(2)
transferring data	read(2), write(2), send(2), recv(2)
addresses	inet(4F)
protocols	tcp(4P), udp(4P).

Acknowledgements

I would like to thank Sam Leffler and Mike Karels for their help in understanding the IPC mechanisms and all the people whose comments have helped in writing and improving this report.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

References

B.W. Kernighan & R. Pike, 1984,
The UNIX Programming Environment.
Englewood Cliffs, N.J.: Prentice-Hall.

B.W. Kernighan & D.M. Ritchie, 1978,
The C Programming Language,
Englewood Cliffs, N.J.: Prentice-Hall.

S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller & C. Torek, 1986,
An Advanced 4.3BSD Interprocess Communication Tutorial.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Computer Systems Research Group, 1986,
UNIX Programmer's Manual, 4.3 Berkeley Software Distribution.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

An Advanced 4.3BSD Interprocess Communication Tutorial

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Phil Lapsley

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Steve Miller

Chris Torek

Heterogeneous Systems Laboratory
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742

PS1 2

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.3BSD release of the UNIX* system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

* UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

One of the most important additions to UNIX in 4.2BSD was interprocess communication. These facilities were the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporated many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. The current release of Berkeley UNIX, 4.3BSD, completes some of the IPC facilities and provides an upward-compatible interface. It is hoped that the interprocess communication facilities included in 4.3BSD will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system, either through naming or implementation. Consequently, the IPC facilities provided in 4.3BSD have been designed as a totally independent subsystem. The 4.3BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

This document provides a high-level description of the IPC facilities in 4.3BSD and their use. It is designed to complement the manual pages for the IPC primitives by examples of their use. The remainder of this document is organized in four sections. Section 2 introduces the IPC-related system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the IPC facilities.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.3BSD IPC facilities support three separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes which communicate using the the DARPA standard communication protocols; and the NS domain, which is used by processes which communicate using the Xerox standard communication protocols*. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the error conditions which are possible when operating in the Internet (or NS) domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Four types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes†.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as part of the NS socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the SPP or IDP headers on a packet or a group of packets either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets. The use of these options is considered in section 5.

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with reliable

* See *Internet Transport Protocols*, Xerox System Integration Standard (XSIS)028112 for more information. This document is almost a necessity for one trying to write NS applications.

† In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to Xerox's Packet Exchange Protocol (PEX) may be simulated at the user level. More information on this topic can be found in section 5.

2.2. Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET; and for the NS domain, AF_NS. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, or SOCK_SEQPACKET must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use the call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in section 5.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding local names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet and NS domains, an association is composed of local and foreign addresses, and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names (the phrase "foreign pathname" means a pathname created by a foreign process, not a pathname on a foreign system). In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local pathname, foreign pathname> tuples. The pathnames may not refer to files already existing on the system in 4.3; the situation may change in future releases.

The *bind* system call allows a process to specify half of an association, <local address, local port> (or <local pathname>), while the *connect* and *accept* primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the *connect* and *send* calls will automatically bind an appropriate address if they are used with an unbound socket. The process of binding names to NS sockets is similar in most ways to that of binding names to Internet sockets.

The *bind* system call is used as follows:

---

* The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). As mentioned, in the Internet domain names contain an Internet address and port number. NS domain names contain an NS address and port number. In the UNIX domain, names contain a path name and a family, which is always AF_UNIX. If one wanted to bind the name "/tmp/foo" to a UNIX domain socket, the following code would be used*:

```
#include <sys/un.h>
...
struct sockaddr_un addr;
...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
  sizeof (addr.sun_family));
```


Note that in determining the size of a UNIX domain address null bytes are not counted, which is why *strlen* is used. In the current implementation of UNIX domain IPC under 4.3BSD, the file name referred to in *addr.sun_path* is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed. Future versions of 4BSD may not create this file.

In binding an Internet address things become more complicated. The actual call is similar,

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

Binding an NS address to a socket is even more difficult, especially since the Internet library routines do not work with NS hostnames. The actual call is again similar:

```
#include <sys/types.h>
#include <netns/ns.h>
...
struct sockaddr_ns sns;
...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

Again, discussion of what to place in a "struct sockaddr_ns" will be deferred to section 3.

2.4. Connection establishment

Connection establishment is usually asymmetric, with one process a "client" and the other a "server". The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively "listens" on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a "connection" to the server's socket. On the client side the *connect* call is used to initiate a connection. Using the

* Note that, although the tendency here is to call the "addr" structure "sun", doing so would cause problems if the code were ever ported to a Sun workstation.

UNIX domain, this might appear as,

```
struct sockaddr_un server;
...
connect(s, (struct sockaddr *)&server, strlen(server.sun_path) +
  sizeof (server.sun_family));
```

while in the Internet domain,

```
struct sockaddr_in server;
...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

and in the NS domain,

```
struct sockaddr_ns server;
...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where *server* in the example above would contain either the UNIX pathname, Internet address and port number, or NS address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary; c.f. section 5.4. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. This is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
struct sockaddr_in from;
...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, and for the NS domain, *from* would be declared as a *struct sockaddr_ns*, but nothing different would need to be done as far as *fromlen* is concerned. In the examples which follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are usable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags, defined in <*sys/socket.h*>, may be specified as a non-zero value if one or more of the following is required:

MSG_OOB	send/receive out of band data
MSG_PEEK	look at data without reading
MSG_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PEEK is specified with a *recv* call, any data present is returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it

may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). *Accept* and *listen* are not used with datagram sockets.

While a datagram socket socket is connected, errors from recent *send* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
#include <sys/time.h>
#include <sys/types.h>
...

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

Select takes as arguments pointers to three sets, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented by the socket If the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the *select* should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array is be long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the *select* call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected; if the *select* call returns due to the timeout expiring, then the value 0 is returned. If the *select* terminates because of an error or interruption, a −1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the *FD_ISSET(fd, &mask)* macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an *accept* call, *select* can be used, followed by a *FD_ISSET(fd, &mask)* macro to check for read readiness on the appropriate socket. If *FD_ISSET* returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

```
#include <sys/time.h>
#include <sys/types.h>
...
fd_set read_template;
struct timeval wait;
...
for (;;) {
        wait.tv_sec = 1;            /* one second */
        wait.tv_usec = 0;

        FD_ZERO(&read_template);

        FD_SET(s1, &read_template);
        FD_SET(s2, &read_template);

        nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
        if (nb <= 0) {
                An error occurred during the select, or
                the select timed out.
        }

        if (FD_ISSET(s1, &read_template)) {
                Socket #1 is ready to be read from.
        }

        if (FD_ISSET(s2, &read_template)) {
                Socket #2 is ready to be read from.
        }
}
```

In 4.2, the arguments to *select* were pointers to integers instead of pointers to *fd_sets*. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.3BSD networking facilities support both the DARPA standard Internet protocols and the Xerox NS protocols, most of the routines presented in this section do not apply to the NS domain. Unless otherwise stated, it should be assumed that the routines presented in this section do not apply to the NS domain.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

3.1. Host names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct    hostent {
              char    *h_name;          /* official name of host */
              char    **h_aliases;      /* alias list */
              int     h_addrtype;       /* host address type (e.g., AF_INET) */
              int     h_length;         /* length of address */
              char    **h_addr_list;    /* list of addresses, null terminated */
};

#define   h_addr  h_addr_list[0]        /* first address, network byte order */
```

The routine *gethostbyname*(3N) takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps Internet host addresses into a *hostent* structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null terminated list of variable length address. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided either by the file */etc/hosts* (*hosts*(5)), or by use of a nameserver, *named*(8). Because of the differences in these databases and their access protocols, the information returned may differ. When using the host table version of *gethostbyname*, only one address will be returned, but all listed aliases will be included. The nameserver version may return alternate addresses, but will not provide any aliases other than one given as argument.

Unlike Internet names, NS names are always mapped into host addresses by the use of a standard NS *Clearinghouse service*, a distributed name and authentication server. The algorithms for mapping NS

names to addresses via a Clearinghouse are rather complicated, and the routines are not part of the standard libraries. The user-contributed Courier (Xerox remote procedure call protocol) compiler contains routines to accomplish this mapping; see the documentation and examples provided therein for more information. It is expected that almost all software that has to communicate using NS will need to use the facilities of the Courier compiler.

An NS host address is represented by the following:

```
union ns_host {
        u_char      c_host[6];
        u_short     s_host[3];
};

union ns_net {
        u_char      c_net[4];
        u_short     s_net[2];
};

struct ns_addr {
        union ns_net        x_net;
        union ns_host       x_host;
        u_short     x_port;
};
```

The following code fragment inserts a known NS address into a *ns_addr*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
...
u_long netnum;
struct sockaddr_ns dst;
...
bzero((char *)&dst, sizeof(dst));

/*
 * There is no convenient way to assign a long
 * integer to a "union ns_net" at present; in
 * the future, something will hopefully be provided,
 * but this is the portable way to go for now.
 * The network number below is the one for the NS net
 * that the desired host (gyre) is on.
 */
netnum = htonl(2266);
dst.sns_addr.x_net = *(union ns_net *) &netnum;
dst.sns_family = AF_NS;

/*
 * host 2.7.1.0.2a.18 == "gyre:Computer Science:UofMaryland"
 */
dst.sns_addr.x_host.c_host[0] = 0x02;
dst.sns_addr.x_host.c_host[1] = 0x07;
dst.sns_addr.x_host.c_host[2] = 0x01;
dst.sns_addr.x_host.c_host[3] = 0x00;
dst.sns_addr.x_host.c_host[4] = 0x2a;
dst.sns_addr.x_host.c_host[5] = 0x18;
dst.sns_addr.x_port = htons(75);
```

PS1 2

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char        *n_name;        /* official name of net */
        char        **n_aliases;    /* alias list */
        int         n_addrtype;     /* net address type */
        int         n_net;          /* network number, host byte order */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above. The routines extract their information from */etc/networks*.

NS network numbers are determined either by asking your local Xerox Network Administrator (and hardcoding the information into your code), or by querying the Clearinghouse for addresses. The internetwork router is the only process that needs to manipulate network numbers on a regular basis; if a process wishes to communicate with a machine, it should ask the Clearinghouse for that machine's address (which will include the net number).

3.3. Protocol names

For protocols, which are defined in */etc/protocols*, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

In the NS domain, protocols are indicated by the "client type" field of a IDP header. No protocol database exists; see section 5 for more information.

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the file */etc/services*. A service mapping is described by the *servent* structure,

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port number, network byte order */
        char    *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

In the NS domain, services are handled by a central dispatcher provided as part of the Courier remote procedure call facilities. Again, the reader is referred to the Courier compiler documentation and to the Xerox standard* for further details.

3.5. Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be

* *Courier: The Remote Procedure Call Protocol*, XSIS 038112.

worthwhile.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, such as the VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines where unneeded these routines are defined as null macros.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
main(argc, argv)
        int argc;
        char *argv[];
{
        struct sockaddr_in server;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&server, sizeof (server));
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        /* Connect does the bind() for us */

        if (connect(s, (char *)&server, sizeof (server)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing whatever appropriate actions the client requests of it.

Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.3BSD, this scheme has been implemented via *inetd*, the so called "internet super-server." *Inetd* listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. *Inetd* will be described in more detail in section 5.

A similar alternative scheme is used by most Xerox services. In general, the Courier dispatch process (if used) accepts connections from processes requesting services of some sort or another. The client processes request a particular <program number, version number, procedure number> triple. If the dispatcher knows of such a program, it is started to handle the request; if not, an error is reported to the client. In this way, only one port is required to service a large variety of different requests. Again, the Courier facilities are not available without the use and installation of the Courier compiler. The information presented in this section applies only to NS clients and services that do not use Courier.

4.1. Servers

In 4.3BSD most servers are accessed at well known Internet addresses or UNIX domain names. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

The result of the *getservbyname* call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

```
main(argc, argv)
        int argc;
        char *argv[];
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        /* Disassociate server from controlling terminal */
        ...
#endif

        sin.sin_port = sp->s_port;  /* Restricted port -- see section 5 */
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, (struct sockaddr *) &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                syslog(LOG_ERR, "rlogind: accept: %m");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

Step two is to disassociate the server from the controlling terminal of its invoker:

```
for (i = 0; i < 3; ++i)
        close(i);

open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
        ioctl(i, TIOCNOTTY, 0);
        close(i);
}
```

This step is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself it can no longer send reports of errors to a terminal, and must log errors via *syslog*.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a "restricted port number" is 4BSD specific, and is covered in section 5.

The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        if (fork() == 0) {  /* Child */
                close(f);
                doit(g, &from);
        }
        close(g);           /* Parent */
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is logged via *syslog* if an error has occurred.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the *accept* is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that *connect* implicitly performs a *bind* call, since *s* is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
...
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are

```
arpa      up   9:45,     5 users, load  1.15,  1.39,  1.31
cad       up   2+12:04,  8 users, load  4.67,  5.13,  4.59
calder    up   10:10,    0 users, load  0.27,  0.15,  0.14
dali      up   2+06:28,  9 users, load  1.04,  1.20,  1.65
degas     up   25+09:48, 0 users, load  1.49,  1.43,  1.41
ear       up   5+00:05,  0 users, load  1.51,  1.54,  1.56
ernie     down 0:24
esvax     down 17:04
ingres    down 0:26
kim       up   3+09:16,  8 users, load  2.03,  2.46,  3.11
matisse   up   3+06:18,  0 users, load  0.03,  0.03,  0.05
medea     up   3+09:39,  2 users, load  0.35,  0.37,  0.50
merlin    down 19+15:37
miro      up   1+07:20,  7 users, load  4.59,  3.28,  2.12
monet     up   1+00:43,  2 users, load  0.22,  0.09,  0.07
oz        down 16:09
statvax   up   2+15:57,  3 users, load  1.52,  1.81,  1.86
ucbvax    up   9:34,     2 users, load  6.08,  5.16,  3.28
```

Figure 3. ruptime output.

interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.3BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information*. A mechanism exists, in the form of an *ioctl* call, for finding the collection of networks to which a host is directly connected. Further, a local

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

* An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        on = 1;
        if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
                syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
                exit(1);
        }
        bind(s, (struct sockaddr *) &sin, sizeof (sin));
        ...
        signal(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                   (struct sockaddr *)&from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                syslog(LOG_ERR, "rwhod: recv: %m");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        syslog(LOG_ERR, "rwhod: %d: bad from port",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        syslog(LOG_ERR, "rwhod: malformed host name from %x",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows 4.3BSD to solve the problem of deciding how to propagate status information in the case of *rwho*, or more generally in broadcasting: Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of such broadcastings are complex, however, and will be covered in section 5.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" (via MSG_PEEK) at out of band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the SIGURG signal via the appropriate *fcntl* call, as described below for SIGIO. If multiple sockets may have out of band data awaiting delivery, a *select* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushs any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out of band message the MSG_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., *telnet*(1C)) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see *setsockopt*(2) for usage. With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ], mark;

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                ...
        }
        ...
}
```

Figure 5. Flushing terminal I/O on receipt of out of band data.

5.2. Non-Blocking Sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the *socket* call, it may be marked as non-blocking by *fcntl* as follows:

```
#include <fcntl.h>
...
int     s;
...
s = socket(AF_INET, SOCK_STREAM, 0);
...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
...
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, *accept*, *connect*, *send*, *recv*, *read*, and *write* can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a *send* cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

5.3. Interrupt driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the *signal* or *sigvec* calls. Second, it must set the process id or process group id which is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This is accomplished by use of an *fcntl* call. Third, it must enable asynchronous notification of pending I/O requests with another *fcntl* call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket *s* is given in Figure 6. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
...
int    io_handler();
...
signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
}
```

Figure 6. Use of asynchronous notification of I/O requests.

5.4. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN *fcntl*, such as was done in the code above for SIGIO. To set the socket's process id for signals, positive arguments should be given to the *fcntl* call. To set the socket's process group for signals, negative arguments should be passed to *fcntl*. Note that the process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar *fcntl*, F_GETOWN, is available for determining the current process number of a socket.

Another signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Figure 2 may be augmented as shown in Figure 7.

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

5.5. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal*. A pseudo- terminal is actually a pair of devices, master

```
int reaper();
 ...
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_ERR, "rlogind: accept: %m");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Figure 7. Use of the SIGCHLD signal.

and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side are processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection— that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out of band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under 4.3BSD, the name of the slave side of a pseudo-terminal is of the form */dev/ttyxy*, where *x* is a single letter starting at 'p' and continuing to 't'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is */dev/ptyxy*, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal which is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then *forks*; the child closes the master side of the pseudo-terminal, and *execs* the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given in Figure 8; this code assumes that a connection on a socket *s* exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from

any previous controlling terminal.

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptyXX";
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;
        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}
if (!gotpty) {
        syslog(LOG_ERR, "All network ports in use");
        exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR); /* slave is now slave side */
if (slave < 0) {
        syslog(LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}

ioctl(slave, TIOCGETP, &b);  /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

i = fork();
if (i < 0) {
        syslog(LOG_ERR, "fork: %m");
        exit(1);
} else if (i) {                 /* Parent */
        close(slave);
        ...
} else {                /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);
        ...
}
```

Figure 8. Creation and use of a pseudo terminal

5.6. Selecting specific protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream based connection, but with protocol type of "newtcp" instead of the default "tcp."

In the NS domain, the available socket protocols are defined in *<netns/ns.h>*. To create a raw socket for Xerox Error Protocol messages, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
...
s = socket(AF_NS, SOCK_RAW, NSPROTO_ERROR);
```

5.7. Address binding

As was mentioned in section 2, binding addresses to sockets in the Internet and NS domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a process may specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests which are addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. This shortcut will work both in the Internet and NS domains. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
        ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that on 4BSD systems, Internet ports below IPPORT_RESERVED (1024) (for the Xerox domain, 0 through 3000) are reserved for privileged users (i.e., the super user); Internet ports above IPPORT_USERRESERVED (50000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the *rresvport* library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
s = rresvport(&lport);
if (s < 0) {
        if (errno == EAGAIN)
                fprintf(stderr, "socket: all ports in use\n");
        else
                perror("rresvport: socket");
        ...
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number. For example, the *rlogin*(1) command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file */etc/hosts.equiv* on the system he is logging in to (or the system name and the user name are in the user's *.rhosts* file in the user's home directory), and second, that the user's rlogin process is coming from a privileged port on the machine from which he is logging. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the *accept* call, or from the *getpeername* call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file

transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
...
int    on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

5.8. Broadcasting and determining network configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

or

```
s = socket(AF_NS, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int    on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

or, for the NS domain,

```
sns.sns_family = AF_NS;
netnum = htonl(net);
sns.sns_addr.x_net = *(union ns_net *) &netnum; /* insert net number */
sns.sns_addr.x_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in *<netinet/in.h>*. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, 4.3BSD provides a method of retrieving this information from the system data structures. The SIOCGIFCONF *ioctl*

call returns the interface configuration of a host in the form of a single *ifconf* structure; this structure contains a "data area" which is made up of an array of of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in *<net/if.h>* as follows:

```
struct ifconf {
        int     ifc_len;                        /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
};

#define ifc_buf  ifc_ifcu.ifcu_buf             /* buffer address */
#define ifc_req  ifc_ifcu.ifcu_req             /* array of structures returned */

#define IFNAMSIZ        16

struct ifreq {
        char    ifr_name[IFNAMSIZ];             /* if name, e.g. "en0" */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                caddr_t ifru_data;
        } ifr_ifru;
};

#define ifr_addr        ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags       ifr_ifru.ifru_flags     /* flags */
#define ifr_data        ifr_ifru.ifru_data      /* for use by interface */
```

The actual call which obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
        ...
}
```

After this call *buf* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

For each structure there exists a set of "interface flags" which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
        /*
         * We must be careful that we don't use an interface
         * devoted to an address family other than those intended;
         * if we were interested in NS interfaces, the
         * AF_INET would be AF_NS.
         */
        if (ifr->ifr_addr.sa_family != AF_INET)
                continue;
        if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
                ...
        }
        /*
         * Skip boring cases.
         */
        if ((ifr->ifr_flags & IFF_UP) == 0 ||
            (ifr->ifr_flags & IFF_LOOPBACK) ||
            (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
                continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR *ioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
        if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_dstaddr, (char *) &dst, sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
        if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_broadaddr, (char *) &dst, sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate *ioctl*'s have obtained the broadcast or destination address (now in *dst*), the *sendto* call may be used:

```
        sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
}
```

In the above loop one *sendto* occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the senders address and port, as datagram sockets are bound before a message is allowed to go out.

5.9. Socket Options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; in most cases this is the "socket level", indicated by the symbolic constant SOL_SOCKET, defined in *<sys/socket.h>*. The actual option is specified in *optname*, and is a symbolic constant also defined in *<sys/socket.h>*. *Optval* and *Optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
        ...
}
```

After the *getsockopt* call, *type* will be set to the value of the socket type, as defined in *<sys/socket.h>*. If, for example, the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

5.10. NS Packet Sequences

The semantics of NS connections demand that the user both be able to look inside the network header associated with any incoming packet and be able to specify what should go in certain fields of an outgoing packet. Using different calls to *setsockopt*, it is possible to indicate whether prototype headers will be associated by the user with each outgoing packet (SO_HEADERS_ON_OUTPUT), to indicate whether the headers received by the system should be delivered to the user (SO_HEADERS_ON_INPUT), or to indicate default information that should be associated with all outgoing packets on a given socket (SO_DEFAULT_HEADERS).

The contents of a SPP header (minus the IDP header) are:

```
struct sphdr {
        u_char  sp_cc;                  /* connection control */
#define SP_SP   0x80                    /* system packet */
#define SP_SA   0x40                    /* send acknowledgement */
#define SP_OB   0x20                    /* attention (out of band data) */
#define SP_EM   0x10                    /* end of message */
        u_char  sp_dt;                  /* datastream type */
        u_short sp_sid;                 /* source connection identifier */
        u_short sp_did;                 /* destination connection identifier */
        u_short sp_seq;                 /* sequence number */
        u_short sp_ack;                 /* acknowledge number */
        u_short sp_alo;                 /* allocation number */
};
```

Here, the items of interest are the *datastream type* and the *connection control* fields. The semantics of the datastream type are defined by the application(s) in question; the value of this field is, by default, zero, but it can be used to indicate things such as Xerox's Bulk Data Transfer Protocol (in which case it is set to one). The connection control field is a mask of the flags defined just below it. The user may set or clear the end-of-message bit to indicate that a given message is the last of a given substream type, or may set/clear the attention bit as an alternate way to indicate that a packet should be sent out-of-band. As an example, to associate prototype headers with outgoing SPP packets, consider:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr_ns sns, to;
int s, on = 1;
struct databuf {
        struct sphdr proto_spp;  /* prototype header */
        char buf[534];           /* max. possible data by Xerox std. */
} buf;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &on, sizeof(on));
 ...
buf.proto_spp.sp_dt = 1; /* bulk data */
buf.proto_spp.sp_cc = SP_EM;        /* end-of-message */
strcpy(buf.buf, "hello world\n");
sendto(s, (char *) &buf, sizeof(struct sphdr) + strlen("hello world\n"),
   (struct sockaddr *) &to, sizeof(to));
 ...
```

Note that one must be careful when writing headers; if the prototype header is not written with the data with which it is to be associated, the kernel will treat the first few bytes of the data as the header, with unpredictable results. To turn off the above association, and to indicate that packet headers received by the system should be passed up to the user, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
...
struct sockaddr sns;
int s, on = 1, off = 0;
...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &off, sizeof(off));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_INPUT, &on, sizeof(on));
...
```

Output is handled somewhat differently in the IDP world. The header of an IDP-level packet looks like:

```
struct idp {
        u_short         idp_sum;        /* Checksum */
        u_short         idp_len;        /* Length, in bytes, including header */
        u_char          idp_tc;         /* Transport Control (i.e., hop count) */
        u_char          idp_pt;         /* Packet Type (i.e., level 2 protocol) */
        struct ns_addr  idp_dna;        /* Destination Network Address */
        struct ns_addr  idp_sna;        /* Source Network Address */
};
```

The primary field of interest in an IDP header is the *packet type* field. The standard values for this field are (as defined in *<netns/ns.h>*):

```
#define NSPROTO_RI      1       /* Routing Information */
#define NSPROTO_ECHO    2       /* Echo Protocol */
#define NSPROTO_ERROR   3       /* Error Protocol */
#define NSPROTO_PE      4       /* Packet Exchange */
#define NSPROTO_SPP     5       /* Sequenced Packet */
```

For SPP connections, the contents of this field are automatically set to NSPROTO_SPP; for IDP packets, this value defaults to zero, which means "unknown".

Setting the value of that field with SO_DEFAULT_HEADERS is easy:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>
 ...
struct sockaddr sns;
struct idp proto_idp;          /* prototype header */
int s, on = 1;
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
proto_idp.idp_pt = NSPROTO_PE;  /* packet exchange */
setsockopt(s, NSPROTO_IDP, SO_DEFAULT_HEADERS, (char *) &proto_idp,
   sizeof(proto_idp));
 ...
```


Using SO_HEADERS_ON_OUTPUT is somewhat more difficult. When SO_HEADERS_ON_OUTPUT is turned on for an IDP socket, the socket becomes (for all intents and purposes) a raw socket. In this case, all the fields of the prototype header (except the length and checksum fields, which are computed by the kernel) must be filled in correctly in order for the socket to send and receive data in a sensible manner. To be more specific, the source address must be set to that of the host sending the data; the destination address must be set to that of the host for whom the data is intended; the packet type must be set to whatever value is desired; and the hopcount must be set to some reasonable value (almost always zero). It should also be noted that simply sending data using *write* will not work unless a *connect* or *sendto* call is used, in spite of the fact that it is the destination address in the prototype header that is used, not the one given in either of those calls. For almost all IDP applications , using SO_DEFAULT_HEADERS is easier and more desirable than writing headers.

5.11. Three-way Handshake

The semantics of SPP connections indicates that a three-way handshake, involving changes in the datastream type, should — but is not absolutely required to — take place before a SPP connection is closed. Almost all SPP connections are "well-behaved" in this manner; when communicating with any process, it is best to assume that the three-way handshake is required unless it is known for certain that it is not required. In a three-way close, the closing process indicates that it wishes to close the connection by sending a zero-length packet with end-of-message set and with datastream type 254. The other side of the connection indicates that it is OK to close by sending a zero-length packet with end-of-message set and datastream type 255. Finally, the closing process replies with a zero-length packet with substream type 255; at this point, the connection is considered closed. The following code fragments are simplified examples of how one might handle this three-way handshake at the user level; in the future, support for this type of close will probably be provided as part of the C library or as part of the kernel. The first code fragment below illustrates how a process might handle three-way handshake if it sees that the process it is communicating with wants to close the connection:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
...
read(s, buf, BUFSIZE);
if (((struct sphdr *)buf)->sp_dt == SPPSST_END) {
        /*
         * SPPSST_END indicates that the other side wants to
         * close.
         */
        proto_sp.sp_dt = SPPSST_ENDREPLY;
        proto_sp.sp_cc = SP_EM;
        setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
           sizeof(proto_sp));
        write(s, buf, 0);
        /*
         * Write a zero-length packet with datastream type = SPPSST_ENDREPLY
         * to indicate that the close is OK with us. The packet that we
         * don't see (because we don't look for it) is another packet
         * from the other side of the connection, with SPPSST_ENDREPLY
         * on it it, too. Once that packet is sent, the connection is
         * considered closed; note that we really ought to retransmit
         * the close for some time if we do not get a reply.
         */
        close(s);
}
...
```

To indicate to another process that we would like to close the connection, the following code would suffice:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
proto_sp.sp_dt = SPPSST_END;
proto_sp.sp_cc = SP_EM;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
write(s, buf, 0);    /* send the end request */
proto_sp.sp_dt = SPPSST_ENDREPLY;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
/*
 * We assume (perhaps unwisely)
 * that the other side will send the
 * ENDREPLY, so we'll just send our final ENDREPLY
 * as if we'd seen theirs already.
 */
write(s, buf, 0);
close(s);
 ...
```


5.12. Packet Exchange

The Xerox standard protocols include a protocol that is both reliable and datagram-oriented. This protocol is known as Packet Exchange (PEX or PE) and, like SPP, is layered on top of IDP. PEX is important for a number of things: Courier remote procedure calls may be expedited through the use of PEX, and many Xerox servers are located by doing a PEX "BroadcastForServers" operation. Although there is no implementation of PEX in the kernel, it may be simulated at the user level with some clever coding and the use of one peculiar *getsockopt*. A PEX packet looks like:

```
/*
 * The packet-exchange header shown here is not defined
 * as part of any of the system include files.
 */
struct pex {
        struct idp  p_idp;                  /* idp header */
        u_short     ph_id[2];               /* unique transaction ID for pex */
        u_short     ph_client;              /* client type field for pex */
};
```

The *ph_id* field is used to hold a "unique id" that is used in duplicate suppression; the *ph_client* field indicates the PEX client type (similar to the packet type field in the IDP header). PEX reliability stems from the fact that it is an idempotent ("I send a packet to you, you send a packet to me") protocol. Processes on each side of the connection may use the unique id to determine if they have seen a given packet before (the unique id field differs on each packet sent) so that duplicates may be detected, and to indicate which message a given packet is in response to. If a packet with a given unique id is sent and no response is received

in a given amount of time, the packet is retransmitted until it is decided that no response will ever be received. To simulate PEX, one must be able to generate unique ids -- something that is hard to do at the user level with any real guarantee that the id is really unique. Therefore, a means (via *getsockopt*) has been provided for getting unique ids from the kernel. The following code fragment indicates how to get a unique id:

```
long uniqueid;
int s, idsize = sizeof(uniqueid);
...
s = socket(AF_NS, SOCK_DGRAM, 0);
...
/* get id from the kernel -- only on IDP sockets */
getsockopt(s, NSPROTO_PE, SO_SEQNO, (char *)&uniqueid, &idsize);
...
```

The retransmission and duplicate suppression code required to simulate PEX fully is left as an exercise for the reader.

5.13. Inetd

One of the daemons provided with 4.3BSD is *inetd*, the so called "internet super-server." *Inetd* is invoked at boot time, and determines from the file */etc/inetd.conf* the servers for which it is to listen. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

Inetd then performs a *select* on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. *Inetd* then performs an *accept* on the socket in question, *forks*, *dups* the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and *execs* the appropriate server.

Servers making use of *inetd* are considerably simplified, as *inetd* takes care of the majority of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as *read*, *write*, *send*, or *recv*. Indeed, servers may use buffered I/O as provided by the "stdio" conventions, as long as as they remember to use *fflush* when appropriate.

One call which may be of interest to individuals writing servers under *inetd* is the *getpeername* call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in "dot notation" (e.g., "128.32.0.4") of a client connected to a server under *inetd*, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
        syslog(LOG_ERR, "getpeername: %m");
        exit(1);
} else
        syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
...
```

While the *getpeername* call is especially useful when writing programs to run with *inetd*, it can be used under other circumstances. Be warned, however, that *getpeername* will fail on UNIX domain sockets.

Debugging with dbx

dbx is an interactive, assembly-level debugger. Although **dbx** is line oriented, the Apollo version has been extended to display source files in its Display Manager window. **dbx** allows you to determine where a program crashes, to view the values of variables and expressions, to set breakpoints in the code, and to run and trace a program. Source code may be in C, Fortran, or Pascal.

Introduction

To use **dbx**, you must compile programs for debugging with full debugging options on, otherwise **dbx** will not find program variables. For example, C programs must be compiled with the **-g** option, as follows:

% **/bin/cc** **-g** *program.c* **-o** *program*

You can run **dbx** on programs compiled without the **-g** option. If you do, however, you can set breakpoints and trace execution, but cannot examine variables. Stripped programs do not have enough symbolic information for debugging.

Invoke the debugger as follows, where *program* is the pathname of the executable file that dumps core:

% **dbx** *program*

The core image should be in the working directory; if it isn't, specify its pathname in the argument after the program name.

Among the advances of **dbx** is that it has a help facility; type **help** to see a list of possible requests. You can obtain help on any **dbx** request by giving its name as an argument to **help**.

Locating a program crash point

dbx is useful for determining what is wrong with a program that crashes. For example, consider the following program, *echo.c*, which indexes past the end of an array:

```
#include <stdio.h>

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

 for (i = 1; i <= argc; ++i)    /* loops once too many */
     puts(argv[i]);
}
```

This program de-references **argv[argc]**, which is usually a NULL pointer. On Apollo systems, de-referencing a NULL pointer causes the program to fault with a segmentation violation. Once this has been observed, the programmer can run the program under **dbx** to find out where in the program the fault happens and to look at the state of variables at the time of the fault.

You can invoke **dbx** as follows, giving the name of the executable file on the **dbx** command line:

```
% dbx a.out
dbx version sr10.2(4) of 9/11/89 16:46 (apollo)
reading symbolic information ...
Type 'help' for help.
(dbx) run spot run
spot
run

Segmentation fault in memccpy at 0x3b4d881c
3b4d881c    MOVE.b      (a0)+,(a1)+
```

dbx reads the executable file and then prompts for a command. Here, we used the **run** command to execute our program under **dbx**. Arguments on the **run** command line become the command line arguments to the program. In this case, the **dbx** command **run spot run** is equivalent to typing **a.out spot run** on the command line.

The program will execute until the fault occurs. Since we invoked the program in **dbx**, the program has not yet exited. We can use **dbx** to determine where and possibly why the fault occurs:

```
(dbx) where
memccpy() at 0x3b4d881c
puts() at 0x3b4e5924
main(argc = 3, argv = 0x3b3c755c), line 8 in "//bah/ech/dbx/doc/echo.c"
```

The **where** command produces a stack trace. The stack trace reveals to us that the program faulted in the system routine **memccpy**() when it was called from **puts**(), which we called from the function **main**() in our program. Since **memccpy**() and **puts**() probably do not contain the bug, we should turn our attention to **main**():

```
(dbx) up
puts() at 0x3b4e5924
(dbx) up
main(argc = 3, argv = 0x3b3c755c), line 8 in "//bah/ech/dbx/doc/echo.c"
```

The **up** command tells **dbx** to look at the next frame up the stack. The next frame represents the function that called the current function. Once we have made **main**() the current frame, the source display will show the function source with a narrow pointing to the line that calls **puts**(). We can now print variables in **main**(). The value of **argv[i]** is suspicious, so we print **i** and **argv[i]**.

```
(dbx) print i
3
(dbx) print argv[i]
(nil)
(dbx) print argv[2]
"run"
```

This tells us that the program has passed a NULL pointer to **puts**(), and that the last element in **argv** is **argv[2]**, not **argv[3]**.

Setting breakpoints

By setting a breakpoint in your program, you can cause its execution to stop at a certain location and examine the state of the program at that location. **dbx** allows you to set a breakpoint at any line in the program. The **stop** command sets a breakpoint. You can set one or more program breakpoints.

After setting your breakpoints, you can execute your program with the **run** command. The program will start execution, and stop if it reaches a breakpoint. To continue from a stopping point, use the **cont** command. Another **run** command would restart the program from the beginning. You can add more breakpoints while the program is stopped, or remove some with the **delete** command. The **status** command lists all the breakpoints you have set.

It is also possible to execute only the next source statement in the program without setting a breakpoint. The **step** command executes the next source statement, and if that statement is a function call, steps into the new

function. The **next** command executes the next source statement, but does not stop inside any new function calls.

This sample **dbx** session illustrates a few of these commands while debugging our *echo.c* program.

```
% dbx a.out
dbx version sr10.2(4) of 9/11/89 16:46 (apollo)
reading symbolic information ...
Type 'help' for help.
(dbx) stop in main
[1] stop in main
(dbx) run for your life
[1] stopped in main at line 7 in file "//bah/ech/dbx/doc/echo.c"
 7       for (i = 1; i <= argc; ++i)
```

The program has executed until the breakpoint in the function **main()**. Now we decide that the statement we are interested in is line 8 and put a breakpoint there.

```
(dbx) stop at 8
[2] stop at "//bah/ech/dbx/doc/echo.c":8
(dbx) c
[2] stopped in main at line 8 in file "//bah/ech/dbx/doc/echo.c"
 8    puts(argv[i]);
```

We have stopped the program at line 8, but we still have a few statements to go until the bug appears. We could use **step** to march forward to that point, but for the sake of illustration, we will use a conditional **stop**.

```
(dbx) stop at 8 if (argv[i] == 0)
[3] if argv+i*4 = 0 { stop } at "//bah/ech/dbx/doc/echo.c":8
(dbx) c
for
[3] stopped in main at line 8 in file "//bah/ech/dbx/doc/echo.c"
 8    puts(argv[i]);
(dbx) print argv[i]
"your"
```

Here we have specified that we want the program to stop at line 8, but only if the value of **argv[i]** is zero (NULL). We continued the program, and it stopped at line 8, but the value of **argv[i]** is not yet zero. The reason is that we neglected to delete the old breakpoint at line 8.

```
(dbx) status
[1] stop in main
[2] stop at "//bah/ech/dbx/doc/echo.c":8
[3] if argv+i*4 = 0 { stop } at "//bah/ech/dbx/doc/echo.c":8
(dbx) delete 2
(dbx) c
your
life
[3] stopped in main at line 8 in file "//bah/ech/dbx/doc/echo.c"
 8    puts(argv[i]);
(dbx) print argv[i]
(nil)
```

Modifying variables

It is often useful to modify the value of program variables mid-execution to determine the outcome of alternative conditions. The **assign** command in **dbx** allows you to assign new values to variables while your program is stopped. For example, the previous breakpoint session could have continued like this:

```
(dbx) assign argv[1] = argv[0]
(dbx) c
a.out
program exited
```

By changing the value of the pointer from NULL to point to a legal string, we sidestepped the bug in this execution of the program. This does not, however, fix the bug or modify the executable in any way.

Expressions

Expressions in **dbx** are similar to those in C, exept that there is a distinction between / (floating-point division) and *div* (integer division), as in Pascal. The table on the following page shows **dbx** requests organized by function.

Groups of *dbx* Requests	
	execution and tracing
`run`	**execute object file**
`cont`	**continue execution from where it stopped**
`trace`	**display tracing information at specified place**
`stop`	**stop execution at specified place**
`status`	**display active *trace* and *stop* requests**
`delete`	**delete specific *trace* or *stop* requests**
`catch`	**start trapping specified signals**
`ignore`	**stop trapping specified signals**
`step`	**execute the next source line, stepping into functions**
`next`	**execute the next source line, even if it's a function**
	displaying data
`print`	**print the value of an expression**
`whatis`	**print the declaration of a given identifier or type**
`which`	**print outer block associated with identifier**
`whereis`	**print all symbols matching identifier**
`assign`	**set the value of a variable**
	function and procedure handling
`where`	**display active procedures and functions on stack**
`down`	**move down the stack towards stopping point**
`up`	**move up the stack towards *main***
`call`	**call the named function or procedure**
`dump`	**display names and values of all local variables**
	accessing source files and directories
`edit`	**invoke an editor on current source file**
`file`	**change current source file**
`func`	**change the current function or procedure**
`list`	**display lines of source code**
`use`	**set directory list to search for source files**
`/.../`	**search down in file to match regular expression**
`?...?`	**search up in file to match regular expression**
	miscellaneous commands
`sh`	**pass command line to the shell**
`alias`	**change *dbx* command name**
`help`	**explain commands**
`source`	**read commands from external file**
`quit`	**exit the debugger**

An Introduction to the Revision Control System

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

The Revision Control System (RCS) manages software libraries. It greatly increases programmer productivity by centralizing and cataloging changes to a software project. This document describes the benefits of using a source code control system. It then gives a tutorial introduction to the use of RCS.

Functions of RCS

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc. It greatly increases programmer productivity by providing the following functions.

1. RCS stores and retrieves multiple revisions of program and other text. Thus, one can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original -- previous revisions remain accessible.
   a. Maintains each module as a tree of revisions.
   b. Project libraries can be organized centrally, decentralized, or any way you like.
   c. RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.
2. RCS maintains a complete history of changes. Thus, one can find out what happened to a module easily and quickly, without having to compare source listings or having to track down colleagues.
   a. RCS performs automatic record keeping.
   b. RCS logs all changes automatically.
   c. RCS guarantees project continuity.
3. RCS manages multiple lines of development.
4. RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, the merging of changes is automatic.
5. RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.
6. RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one change will not wipe out the other one.
7. RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
8. RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.
9. RCS performs automatic identification of modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules make up a

given configuration.

10. Provides high-level management visibility. Thus, it is easy to track the status of a software project.
    a. RCS provides a complete change history.
    b. RCS records who did what when to which revision of which module.
11. RCS is fully compatible with existing software development tools. RCS is unobtrusive -- its interface to the file system is such that all your existing software tools can be used as before.
12. RCS' basic user interface is extremely simple. The novice only needs to learn two commands. Its more sophisticated features have been tuned towards advanced software development environments and the experienced software professional.
13. RCS simplifies software distribution if customers also maintain sources with RCS. This technique assures proper identification of versions and configurations, and tracking of customer changes. Customer changes can be merged into distributed versions locally or by the development group.
14. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

Getting Started with RCS

Suppose you have a file f.c that you wish to put under control of RCS. Invoke the checkin command:

```
ci f.c
```

This command creates f.c,v, stores f.c into it as revision 1.1, and deletes f.c. It also asks you for a description. The description should be a synopsis of the contents of the file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in ,v are called RCS files ("v" stands for "versions"), the others are called working files. To get back the working file f.c in the previous example, use the checkout command:

```
co f.c
```

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it in back in by invoking:

```
ci f.c
```

Ci increments the revision number properly. If *ci* complains with the message

```
ci error: no lock set by <your login>
```

then your system administrator has decided to create all RCS files with the locking attribute set to "strict". With strict locking, you you must lock the revision during the previous checkout. Thus, your last checkout should have been

```
co -l f.c
```

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Of course, it is too late now to do the checkout with locking, because you probably modified f.c already, and a second checkout would overwrite your changes. Instead, invoke

```
rcs -l f.c
```

This command will lock the latest revision for you, unless somebody else got ahead of you already. If someone else has the lock you will have to negotiate your changes with them.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner off the RCS file need not have a lock for checkin; all others still do. Turning strict locking off and on is done with the commands:

```
rcs -U f.c     and     rcs -L f.c
```

You can set the locking to strict or non-strict on every RCS file.

If you do not want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any change*.

To avoid the deletion of the working file during checkin (should you want to continue editing), invoke

```
ci -l f.c
```

This command checks in f.c as usual, but performs an additional checkout with locking. Thus, it saves you one checkout operation. There is also an option **-u** for *ci* that does a checkin followed by a checkout without locking. This is useful if you want to compile the file after the checkin. Both options also update the identification markers in your file (see below).

You can give *ci* the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The command

```
ci -r2 f.c     or     ci -r2.1 f.c
```

assigns the number 2.1 to the new revision. From then on, *ci* will number the subsequent revisions with 2.2, 2.3, etc. The corresponding *co* commands

```
co -r2 f.c     and     co -r2.1 f.c
```

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

```
ci -r1.3.1 f.c
```

This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision. For more information about branches, see *rcsfile*(5).

Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

```
$Header$
```

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

```
$Header: filename revisionnumber date time author state $
```

You never need to touch this string, because RCS keeps it up to date automatically. To propagate the marker into your object code, simply put it into a literal character string. In C, this is done as follows:

```
static char rcsid[] = "$Header$";
```

The command *ident* extracts such markers from any file, even object code. Thus, *ident* helps you to find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker

```
$Log$
```

into your text, inside a comment. This marker accumulates the log messages that are requested during checkin. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see *co* (1) for details.

* Pairs of RCS and working files can really be specified in 3 ways: a) both are given, b) only the working file is given, c) only the RCS file is given. Both files may have arbitrary path prefixes; RCS commands pair them up intelligently.

How to combine MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule into your makefile. Do not use a rule of the form .c,v.c, because such a rule keeps a copy of every working file checked out, even those you are not working on. Instead, use this:

```
.SUFFIXES: .c,v

.c,v.o:
        co -q $*.c
        cc $(CFLAGS) -c $*.c
        rm -f $*.c

prog:  f1.o f2.o .....
        cc f1.o f2.o ..... -o prog
```

This rule has the following effect. If a file f.c does not exist, and f.o is older than f.c,v, MAKE checks out f.c, compiles f.c into f.o, and then deletes f.c. From then on, MAKE will use f.o until you change f.c,v.

If f.c exists (presumably because you are working on it), the default rule .c.o takes precedence, and f.c is compiled into f.o, but not deleted.

If you keep your RCS file in the directory ./RCS, all this will not work and you have to write explicit checkout rules for every file, like

```
f1.c:  RCS/f1.c,v; co -q f1.c
```

Unfortunately, these rules do not have the property of removing unneeded .c-files.

Additional Information on RCS

If you want to know more about RCS, for example how to work with a tree of revisions and how to use symbolic revision numbers, read the following paper:

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

Taking a look at the manual page *RCSFILE*(5) should also help to understand the revision tree permitted by RCS.

An Introduction to the Source Code Control System

Eric Allman
Project Ingres
University of California at Berkeley

This document gives a quick introduction to using the Source Code Control System (SCCS). The presentation is geared to programmers who are more concerned with what to do to get a task done rather than how it works; for this reason some of the examples are not well explained. For details of what the magic options do, see the section on "Further Information".

This is a working document. Please send any comments or suggestions to eric@Berkeley.Edu.

1. Introduction

SCCS is a source management system. Such a system maintains a record of versions of a system; a record is kept with each set of changes of what the changes are, why they were made, and who made them and when. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, SCCS will insure that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the "s-file". There are three major operations that can be performed on the s-file:

(1) Get a file for compilation (not for editing). This operation retrieves a version of the file from the s-file. By default, the latest version is retrieved. This file is intended for compilation, printing, or whatever; it is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.

(2) Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a file at one time.

(3) Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

2. Learning the Lingo

There are a number of terms that are worth learning before we go any farther.

2.1. S-file

The s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; *i.e.*, only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person who created the version explaining why the changes were made.

2.2. Deltas

Each set of changes to the s-file (which is approximately [but not exactly!] equivalent to a version of the file) is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before[1]. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes — equivalent to removing your changes later.

2.3. SID's (or, version numbers)

A SID (SCCS Id) is a number that represents a delta. This is normally a two-part number consisting of a "release" number and a "level" number. Normally the release number stays the same, however, it is possible to move into a new release if some major change is being made.

Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

2.4. Id keywords

When you get a version of a file with intent to compile and install it (*i.e.*, something other than edit it), some special keywords are expanded inline by SCCS. These *Id Keywords* can be used to include the current version number or other information into the file. All id keywords are of the form **%*x*%**, where *x* is an upper case letter. For example, **%I%** is the SID of the latest delta applied, **%W%** includes the module name, SID, and a mark that makes it findable by a program, and **%G%** is the date of the latest delta applied. There are many others, most of which are of dubious usefulness.

When you get a file for editing, the id keywords are not expanded; this is so that after you put them back in to the s-file, they will be expanded automatically on each new version. But notice: if you were to get them expanded accidently, then your file would appear to be the same version forever more, which would of course defeat the purpose. Also, if you should install a version of the program without expanding the id keywords, it will be impossible to tell what version it is (since all it will have is "%W%" or whatever).

3. Creating SCCS Files

To put source files into SCCS format, run the following shell script from csh:

```
mkdir SCCS save
foreach i (*.[ch])
        sccs admin -i$i $i
        mv $i save/$i
end
```

This will put the named files into s-files in the subdirectory "SCCS" The files will be removed from the current directory and hidden away in the directory "save", so the next thing you will probably want to do is to get all the files (described below). When you are convinced that SCCS has correctly created the s-files, you should remove the directory "save".

If you want to have id keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* will print "No Id Keywords (cm7)", which is a warning message only.

4. Getting Files for Compilation

To get a copy of the latest version of a file, run

```
sccs get prog.c
```

SCCS will respond:

---

[1]This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history.

```
1.1
87 lines
```

meaning that version 1.1 was retrieved[2] and that it has 87 lines. The file *prog.c* will be created in the current directory. The file will be read-only to remind you that you are not supposed to change it.

This copy of the file should not be changed, since SCCS is unable to merge the changes back into the s-file. If you do make changes, they will be lost the next time someone does a *get*.

5. Changing Files (or, Creating Deltas)

5.1. Getting a copy to edit

To edit a source file, you must first get it, requesting permission to edit it[3]:

```
sccs edit prog.c
```

The response will be the same as with *get* except that it will also say:

```
New delta 1.2
```

You then edit it, using a standard text editor:

```
vi prog.c
```

5.2. Merging the changes back into the s-file

When the desired changes are made, you can put your changes into the SCCS file using the *delta* command:

```
sccs delta prog.c
```

Delta will prompt you for "comments?" before it merges the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash[4]). *Delta* will then type:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[5]. The *prog.c* file will be removed; it can be retrieved using *get*.

5.3. When to make deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like "fixed compilation problem in previous delta" or "fixed botch in 1.3". However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get them, and recompile everything.

---

[2]Actually, the SID of the final delta applied was 1.1.

[3]The "edit" command is equivalent to using the –e flag to *get*, as:

```
sccs get –e prog.c
```

Keep this in mind when reading other documentation.

[4]Yes, this is a stupid default.

[5]Changes to a line are counted as a line deleted and a line inserted.

5.4. What's going on: the info command

To find out what files where being edited, you can use:

```
sccs info
```

to print out all the files being edited and other information such as the name of the user who did the edit. Also, the command:

```
sccs check
```

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns non-zero exit status if anything is being edited; it can be used in an "install" entry in a makefile to abort the install if anything has not been properly deltaed.

If you know that everything being edited should be deltaed, you can use:

```
sccs delta `sccs tell`
```

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a **-b** flag to ignore "branches" (alternate versions, described later) and the **-u** flag to only give files being edited by you. The **-u** flag takes an optional *user* argument, giving only files being edited by that user. For example,

```
sccs info -ujohn
```

gives a listing of files being edited by john.

5.5. ID keywords

Id keywords can be inserted into your file that will be expanded automatically by *get*. For example, a line such as:

```
static char SccsId[] = "%W%\t%G%";
```

will be replaced with something like:

```
static char SccsId[] = "@(#)prog.c 1.2	08/29/80";
```

This tells you the name and version of the source file and the time the delta was created. The string "@(#)" is a special string which signals the beginning of an SCCS Id keyword.

5.5.1. The what command

To find out what version of a program is being run, use:

```
sccs what prog.c /usr/bin/prog
```

which will print all strings it finds that begin with "@(#)". This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
        prog.c  1.2     08/29/80
/usr/bin/prog:
        prog.c  1.1     02/05/79
```

From this I can see that the source that I have in prog.c will not compile into the same version as the binary in /usr/bin/prog.

5.5.2. Where to put id keywords

ID keywords can be inserted anywhere, including in comments, but Id Keywords that are compiled into the object module are especially useful, since it lets you find out what version of the object is being run, as well as the source. However, there is a cost: data space is used up to store the keywords, and on small address space machines this may be prohibitive.

When you put id keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W%  %G%";
```

in the file *access.h* and:

```
static char OpsysSid[] = "%W%    %G%";
```

in the file *opsys.h*. Otherwise, you will get compilation errors because "SccsId" is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put id keywords in header files in comments.

5.6. Keeping SID's consistent across files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always *edit* all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are redeltaed. This can be done fairly easily by just specifying the name of the directory that the SCCS files are in:

```
sccs edit SCCS
```

which will *edit* all files in that directory. To make the delta, use:

```
sccs delta SCCS
```

You will be prompted for comments only once.

5.7. Creating new releases

When you want to create a new release of a program, you can specify the release number you want to create on the *edit* command. For example:

```
sccs edit -r2 prog.c
```

will cause the next delta to be in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

```
sccs edit -r2 SCCS
```

6. Restoring Old Versions

6.1. Reverting to old versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3. But this introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

```
sccs get -r1.2 prog.c
```

This will produce a version of *prog.c* that is delta 1.2 that can be reinstalled so that work can proceed.

In some cases you don't know what the SID of the delta you want is. However, you can revert to the version of the program that was running as of a certain date by using the -c (cutoff) flag. For example,

```
sccs get -c800722120000 prog.c
```

will retrieve whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

```
sccs get -c"80/07/22 12:00:00" prog.c
```

6.2. Selectively deleting old deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
sccs edit -x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
sccs edit -x1.3-1.4 prog.c
```

which will exclude all deltas from 1.3 to 1.4. Alternatively,

```
sccs edit -x1.3-1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using **-x** (or **-i**; see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS always prints out a message telling the range of lines effected; these lines should then be examined very carefully to see if the version SCCS got is ok.

Since each delta (in the sense of "a set of changes") can be excluded at will, that this makes it most useful to put each semantically distinct change into its own delta.

7. Auditing Changes

7.1. The prs command

When you created a delta, you presumably gave a reason for the delta to the "comments?" prompt. To print out these comments later, use:

```
sccs prs prog.c
```

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
D 1.2 80/08/29 00:19:31 bill 2 1   00005/00003/00084
MRs:
COMMENTS:
removed "-q" option

D 1.1 79/02/05 00:19:31  eric 1 0 00087/00000/00000
MRs:
COMMENTS:
date and time created 80/06/10 00:19:31 by eric
```

7.2. Finding why lines were inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
sccs get -m prog.c
```

You can then find out what this delta did by printing the comments using *prs*.

To find out what lines are associated with a particular delta (*e.g.*, 1.3), use:

```
sccs get -m -p prog.c | grep ^1.3'
```

The **-p** flag causes SCCS to output the generated source to the standard output rather than to a file.

7.3. Finding what changes you have made

When you are editing a file, you can find out what changes you have made using:

```
sccs diffs prog.c
```

Most of the "diff" flags can be used. To pass the **-c** flag, use **-C**.

To compare two versions that are in deltas, use:

```
sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6.

8. Shorthand Notations

There are several sequences of commands that get executed frequently. *Sccs* tries to make it easy to do these.

8.1. Delget

A frequent requirement is to make a delta of some file and then get that file. This can be done by using:

```
sccs delget prog.c
```

which is entirely equivalent to using:

```
sccs delta prog.c
sccs get prog.c
```

The "deledit" command is equivalent to "delget" except that the "edit" command is used instead of the "get" command.

8.2. Fix

Frequently, there are small bugs in deltas, e.g., compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
sccs fix -r1.4 prog.c
```

This will get a copy of delta 1.4 of prog.c for you to edit and then delete delta 1.4 from the SCCS file. When you do a delta of prog.c, it will be delta 1.4 again. The -r flag must be specified, and the delta that is specified must be a leaf delta, i.e., no other deltas may have been made subsequent to the creation of that delta.

8.3. Unedit

If you found you edited a file that you did not want to edit, you can back out by using:

```
sccs unedit prog.c
```

8.4. The -d flag

If you are working on a project where the SCCS code is in a directory somewhere, you may be able to simplify things by using a shell alias. For example, the alias:

```
alias syssccs sccs -d/usr/src
```

will allow you to issue commands such as:

```
syssccs edit cmd/who.c
```

which will look for the file "/usr/src/cmd/SCCS/who.c". The file "who.c" will always be created in your current directory regardless of the value of the -d flag.

9. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an s-file while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a good scenario for working might be:

```
sccs edit a.c g.c t.c
vi a.c g.c t.c
# do testing of the (experimental) version
sccs delget a.c g.c t.c
sccs info
# should respond "Nothing being edited"
make install
```

As a general rule, all source files should be deltaed before installing the program for general use. This will insure that it is possible to restore any version in use at any time.

10. Saving Yourself

10.1. Recovering a munged edit file

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[6]. Unfortunately, you can't just remove it and re-*edit* it; SCCS keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Neither can you just get it using *get*, since that would expand the Id keywords. Instead, you can say:

```
sccs get -k prog.c
```

This will not expand the Id keywords, so it is safe to do a delta with it.

Alternately, you can *unedit* and *edit* the file.

10.2. Restoring the s-file

In particularly bad circumstances, the SCCS file itself may get munged. The most common way this happens is that it gets edited. Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

```
sccs admin -z prog.c
```

11. Using the Admin Command

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the **-f** flag. For example:

```
sccs admin -fd1 prog.c
```

sets the "d" flag to the value "1". This flag can be deleted by using:

```
sccs admin -dd prog.c
```

The most useful flags are:

b Allow branches to be made using the -b flag to *edit*.

d*SID* Default SID to be used on a *get* or *edit*. If this is just a release number it constrains the version to a particular release only.

i Give a fatal error if there are no Id Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the Id Keywords inserted as constants instead of internal forms.

y The "type" of the module. Actually, the value of this flag is unused by SCCS except that it replaces the **%Y%** keyword.

The -t*file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the **-t** flag insures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted.

---

[6] Or given up and decided to start over.

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

12. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a "branch." Normally deltas continue in a straight line, each depending on the delta before. Creating a branch "forks off" a version of the program.

The ability to create branches must be enabled in advance using:

```
sccs admin -fb prog.c
```

The **-fb** flag can be specified when the SCCS file is first created.

12.1. Creating a branch

To create a branch, use:

```
sccs edit -b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

12.2. Getting from a branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
sccs get -r1.5.1 prog.c
```

12.3. Merging a branch back into the main trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime someone may have created a delta 1.6 that you don't want to lose. The commands:

```
sccs edit -i1.5.1.1-1.5.1 prog.c
sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, get will print an error; the generated result should be carefully examined before the delta is made.

12.4. A more detailed example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
mkdir ../newxyz
cd ../newxyz
```

Edit a copy of the program on a branch:

```
sccs -d../xyz edit prog.c
```

When using the old version, be sure to use the **-b** flag to info, check, tell, and clean to avoid confusion. For example, use:

```
sccs info -b
```

when in the directory "xyz".

If you want to save a copy of the program (still on the branch) back in the s-file, you can use:

```
sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the s-file using delta:

```
sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk (*i.e.* the default version), which may have undergone changes. If so, it can be merged using the **–i** flag to *edit* as described above.

12.5. A warning

Branches should be kept to a minimum. After the first branch from the trunk, SID's are assigned rather haphazardly, and the structure gets complex fast.

13. Using SCCS with Make

SCCS and make can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few basic entries that every makefile ought to have. These are:

a.out	(or whatever the makefile generates.) This entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates many things, this should be called "all" and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special *chmod*'s or *ranlib*'s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes all files from the current directory that can be regenerated from SCCS files.
print	Prints the contents of the directory.

The examples shown below are only partial examples, and may omit some of these entries when they are deemed to be obvious.

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed. To do this, the command:

```
sccs clean
```

can be used. This will remove all files for which an s-file exists, but which is not being edited.

13.1. To maintain single programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= –i –s

prog: prog.o
        $(CC) $(LDFLAGS) –o prog prog.o
prog.o: prog.c prog.h

example: example.o
        $(CC) $(LDFLAGS) –o example example.o
example.o: example.c

.DEFAULT:
        sccs get $<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the **.o** file on the **.c** file is important. Another way of doing the same thing is:

```
SRCS= prog.c prog.h example.c
LDFLAGS= -i -s
prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h
example: example.o
	$(CC) $(LDFLAGS) -o example example.o
sources: $(SRCS)
$(SRCS):
	sccs get $@
```

There are a couple of advantages to this approach: (1) the explicit dependencies of the .o on the .c files are not needed, (2) there is an entry called "sources" so if you want to get all the sources you can just say "make sources", and (3) the makefile is less likely to do confusing things since it won't try to *get* things that do not exist.

13.2. To maintain a library

Libraries that are largely static are best updated using explicit commands, since *make* doesn't know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
# configuration information
OBJS= a.o b.o c.o d.o
SRCS= a.c b.c c.c d.s x.h y.h z.h
TARG= /usr/lib
# programs
GET= sccs get
REL=
AR= -ar
RANLIB= ranlib
lib.a: $(OBJS)
	$(AR) rvu lib.a $(OBJS)
	$(RANLIB) lib.a
install: lib.a
	sccs check
	cp lib.a $(TARG)/lib.a
	$(RANLIB) $(TARG)/lib.a
sources: $(SRCS)
$(SRCS):
	$(GET) $(REL) $@
print: sources
	pr *.h *.[cs]
clean:
	rm -f *.o
	rm -f core a.out $(LIB)
```

The "$(REL)" in the get can be used to get old versions easily; for example:

```
make b.o REL=-r1.3
```

The *install* entry includes the line "sccs check" before anything else. This guarantees that all the s-files are up to date (*i.e.*, nothing is being edited), and will abort the *make* if this condition is not met.

13.3. To maintain a large program

```
OBJS=  a.o b.o c.o d.o
SRCS=  a.c b.c c.y d.s x.h y.h z.h
GET=   sccs get
REL=
a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)
sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@
```

(The *print* and *clean* entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the mod date of z.h in line with the mod date of x.h. When you have a makefile such as above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on z.h.

14. Further Information

The *SCCS/PWB User's Manual* gives a deeper description of how to use SCCS. Of particular interest are the numbering of branches, the l-file, which gives a description of what deltas were used on a get, and certain other SCCS commands.

The SCCS manual pages are a good last resort. These should be read by software managers and by people who want to know everything about everything.

Both of these documents were written without the *sccs* front end in mind, so most of the examples are slightly different from those in this document.

Quick Reference

1. Commands

The following commands should all be preceded with "sccs". This list is not exhaustive; for more options see *Further Information*.

get — Gets files for compilation (not for editing). Id keywords are expanded.

- –r*SID* Version to get.
- –p Send to standard output rather than to the actual file.
- –k Don't expand id keywords.
- –i*list* List of deltas to include.
- –x*list* List of deltas to exclude.
- –m Precede each line with SID of creating delta.
- –c*date* Don't apply any deltas created after *date*.

edit — Gets files for editing. Id keywords are not expanded. Should be matched with a *delta* command.

- –r*SID* Same as *get*. If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID*.
- –b Create a branch.
- –i*list* Same as *get*.
- –x*list* Same as *get*.

delta — Merge a file gotten using *edit* back into the s-file. Collect comments about why this delta was made.

unedit — Remove a file that has been edited previously without merging the changes into the s-file.

prs — Produce a report of changes.

info — Give a list of all files being edited.

- –b Ignore branches.
- –u[*user*] Ignore files not being edited by *user*.

check — Same as *info*, except that nothing is printed if nothing is being edited and exit status is returned.

tell — Same as *info*, except that one line is produced per file being edited containing only the file name.

clean — Remove all files that can be regenerated from the s-file.

what — Find and print id keywords.

admin — Create or set parameters on s-files.

- –i*file* Create, using *file* as the initial contents.
- –z Rebuild the checksum in case the file has been trashed.
- –f*flag* Turn on the *flag*.
- –d*flag* Turn off (delete) the *flag*.
- –t*file* Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or "design & implementation" documents to insure they get distributed with the s-file.

Useful flags are:

	b	Allow branches to be made using the -b flag to *edit*.
	d*SID*	Default SID to be used on a *get* or *edit*.
	i	Cause "No Id Keywords" error message to be a fatal error rather than a warning.
	t	The module "type"; the value of this flag replaces the **%Y%** keyword.
fix	Remove a delta and reedit it.	
delget	Do a *delta* followed by a *get*.	
deledit	Do a *delta* followed by an *edit*.	

2. Id Keywords

%Z% Expands to "@(#)" for the *what* command to find.

%M% The current module name, *e.g.*, "prog.c".

%I% The highest SID applied.

%W% A shorthand for "%Z%%M% <tab> %I%".

%G% The date of the delta corresponding to the "%I%" keyword.

%R% The current release number, *i.e.*, the first component of the "%I%" keyword.

%Y% Replaced by the value of the t flag (set by *admin*).

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that

the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    ...

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere. Yacc has been extensively used in numerous practical applications, including *lint*, the Portable C Compiler, and a system for typesetting mathematics.

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and

Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* ... */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'   newline
'\r'   return
'\''   single quote "'"
'\\'   backslash "\"
'\t'   tab
'\b'   backspace
'\f'   form feed
'\xxx' "xxx" in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar.

PS1 6

Thus the grammar rules

```
A   :   B C D ;
A   :   E F ;
A   :   G ;
```

can be given to Yacc as

```
A   :   B C D
    |   E F
    |   G
    ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :  ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token  name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A   :   '(' B ')'
            {   hello( 1, "abc" ); }
```

and

```
XXX :   YYY ZZZ
            {   printf("a message\n");
                flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components to the right side of a rule, reading from left to right. Thus, if the rule is

```
A    :    B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr :    '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr :    '(' expr ')'          { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A    :    B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A    :    B
              { $$ = 1; }
          C
              { x = $2;  y = $3; }
     ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT      :    /* empty */
               { $$ = 1; }
     ;

A    :    B $ACT C
               { x = $2;  y = $3; }
     ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr :    expr '+' expr
              { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch( c ) {
        ...
    case '0':
    case '1':
     ...
    case '9':
        yylval = c-'0';
        return( DIGIT );
        ...
        }
    ...
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk. These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.       reduce 18
```

refers to *grammar rule* 18, while the action

```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

PS1 6

```
A : x y z ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the end-marker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme   :   sound place
        ;
sound:  DING DONG
        ;
place :  DELL
        ;
```

When Yacc is invoked with the –v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_  (1)

        . reduce 1

state 5
        place : DELL_  (3)

        . reduce 3

state 6
        sound : DING DONG_  (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token

DONG is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr : expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift/reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce/reduce conflict*. Note that there are never any "shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat  :   IF '(' cond ')' stat
      |   IF '(' cond ')' stat ELSE stat
      ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
     IF ( C2 ) S1
     }
ELSE S2
```

or

```
IF ( C1 ) {
    IF ( C2 ) S1
    ELSE S2
    }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

    stat : IF ( cond ) stat_        (18)
    stat : IF ( cond ) stat_ELSE stat

    ELSE    shift 45
            reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

IF (cond) stat

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

stat : IF (cond) stat ELSE_stat

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

stat : IF '(' cond ')' stat

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most one reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

expr : expr OP expr

and

expr : UNARY expr

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

A .LT. B .LT. C

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr :    expr '=' expr
     |    expr '+' expr
     |    expr '-' expr
     |    expr '*' expr
     |    expr '/' expr
     |    NAME
     ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr :    expr '+' expr
     |    expr '-' expr
     |    expr '*' expr
     |    expr '/' expr
     |    '-' expr    %prec '*'
     |    NAME
     ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated

with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :    error  ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input :    error '\n' { printf( "Reenter last line: " ); } input
                  {     $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input :     error '\n'
                { yyerrok;
                  printf( "Reenter last line: " ); }
            input
                { $$ = $4; }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat :      error
                { resynch();
                  yyerrok ;
                  yyclearin ; }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse* ; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex* , the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse* . In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror* . The name of this library is system dependent; on many systems the library is accessed by a –ly argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
    return( yyparse() );
    }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name :      name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list ',' item
        ;
```

and

```
seq     :       item
        |       seq item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for

the second and all succeeding items.

With right recursive rules, such as

```
seq  :  item
     |  item seq
     ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq  :  /* empty */
     |  seq item
     ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
 ... other declarations ...

%%

prog :  decls stats
     ;

decls:  /* empty */
                {  dflag = 1; }
     |  decls declaration
     ;

stats:  /* empty */
                {  dflag = 0; }
     |  stats statement
     ;

 ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved* ; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent  :     adj noun verb adj noun
                  { look at the sentence . . . }
      ;

adj   :     THE         {     $$ = THE; }
      |     YOUNG       {     $$ = YOUNG; }
      . . .
      ;

noun  :     DOG
                  {     $$ = DOG; }
      |     CRONE
                  {     if( $0 == YOUNG ){
                              printf( "what?\n" );
                              }
                        $$ = CRONE;
                        }
      ;
      . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint* will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **–d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule  :     aaa  {  $<intval>$ = 3;  } bbb
                 {      fun( $<intval>2, $<other>0 );  }
      ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*' s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS    /* supplies precedence for unary minus */

%%   /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;

expr    :       '(' expr ')'
                        {       $$ = $2; }
        |       expr '+' expr
                        {       $$ = $1 + $3; }
        |       expr '-' expr
                        {       $$ = $1 - $3; }
        |       expr '*' expr
                        {       $$ = $1 * $3; }
```

```
    | expr '/' expr
                { $$ = $1 / $3; }
      | expr '%' expr
                { $$ = $1 % $3; }
      | expr '&' expr
                { $$ = $1 & $3; }
      | expr '|' expr
                { $$ = $1 | $3; }
      | '-' expr    %prec UMINUS
                { $$ = - $2; }
      | LETTER
                { $$ = regs[$1]; }
      | number
      ;

number :  DIGIT
                { $$ = $1;  base = ($1==0) ? 8 : 10; }
      | number DIGIT
                { $$ = base * $1 + $2; }
      ;

%%   /* start of programs */

yylex() {               /* lexical analysis routine */
      /* returns LETTER for a lower case letter, yylval = 0 through 25 */
      /* return DIGIT for a digit, yylval = 0 through 9 */
      /* all other characters are returned immediately */

      int c;

      while( (c=getchar()) == ' ' ) {   /* skip blanks */ }

      /* c is now nonblank */

      if( islower( c ) ) {
            yylval = c - 'a';
            return ( LETTER );
            }
      if( isdigit( c ) ) {
            yylval = c - '0';
            return( DIGIT );
            }
      return( c );
      }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
/* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /*  includes identifiers and literals */
%token  C_IDENTIFIER    /*  identifier (but not literal) followed by colon  */
%token  NUMBER          /*  [0-9]+  */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK    /* the %% mark */
%token  LCURL   /* the %{ mark */
%token  RCURL   /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK    { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       defs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
```

```
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 − 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
      double lo, hi;
      } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union  {
      int ival;
      double dval;
      INTERVAL vval;
      }

%token <ival> DREG VREG      /* indices into dreg, vreg arrays */

%token <dval> CONST          /* floating point constant */

%type <dval> dexp          /* expression */

%type <vval> vexp          /* interval expression */

     /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS     /* precedence for unary minus */

%%

lines :     /* empty */
      |     lines line
      ;

line  :     dexp '\n'
                  {     printf( "%15.8f\n", $1 ); }
      |     vexp '\n'
                  {     printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
      |     DREG '=' dexp '\n'
                  {     dreg[$1] = $3; }
      |     VREG '=' vexp '\n'
```

```
                { vreg[$1] = $3; }
        |   error '\n'
                { yyerrok; }
        ;

dexp :  CONST
        |   DREG
                { $$ = dreg[$1]; }
        |   dexp '+' dexp
                { $$ = $1 + $3; }
        |   dexp '-' dexp
                { $$ = $1 - $3; }
        |   dexp '*' dexp
                { $$ = $1 * $3; }
        |   dexp '/' dexp
                { $$ = $1 / $3; }
        |   '-' dexp    %prec UMINUS
                { $$ = -$2; }
        |   '(' dexp ')'
                { $$ = $2; }
        ;

vexp :  dexp
                { $$.hi = $$.lo = $1; }
        |   '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        |   VREG
                { $$ = vreg[$1]; }
        |   vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
        |   dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
        |   vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi; }
        |   dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi; }
        |   vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        |   dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        |   vexp '/' vexp
                { if( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
    |   dexp '/' vexp
            {   if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1, $1, $3 ); }
    |   '-' vexp   %prec UMINUS
            {   $$.hi = -$2.lo;  $$.lo = -$2.hi;  }
    |   '(' vexp ')'
            {   $$ = $2; }
    ;

%%

# define BSZ 50     /* buffer size for floating point numbers */

    /* lexical analysis */

yylex(){
    register c;

    while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

    if( isupper( c ) ){
        yylval.ival = c - 'A';
        return( VREG );
        }
    if( islower( c ) ){
        yylval.ival = c - 'a';
        return( DREG );
        }

    if( isdigit( c ) || c=='.' ){
        /* gobble up digits, points, exponents */

        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

            *cp = c;
            if( isdigit( c ) ) continue;
            if( c == '.' ){
                if( dot++ || exp ) return( '.' );  /* will cause syntax error */
                continue;
                }

            if( c == 'e' ){
                if( exp++ ) return( 'e' );  /* will cause syntax error */
                continue;
                }

            /* end of number */
            break;
            }
        *cp = '\0';
        if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

PS1 6

```
        else ungetc( c, stdin );  /* push back last char read */
        yylval.dval = atof( buf );
        return( CONST );
        }
    return( c );
    }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
    /* returns the smallest interval containing a, b, c, and d */
    /* used by *, / routines */
    INTERVAL v;

    if( a>b ) { v.hi = a;  v.lo = b; }
    else { v.hi = b;  v.lo = a; }

    if( c>d ) {
        if( c>v.hi ) v.hi = c;
        if( d<v.lo ) v.lo = d;
        }
    else {
        if( d>v.hi ) v.hi = d;
        if( c<v.lo ) v.lo = c;
        }
    return( v );
    }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
    }

dcheck( v ) INTERVAL v; {
    if( v.hi >= 0. && v.lo <= 0. ){
        printf( "divisor interval contains 0.\n" );
        return( 1 );
        }
    return( 0 );
    }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
    }
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes ‘‘"’’.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash ‘‘\’’ may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec
5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.
6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Lex – A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

1. Introduction.

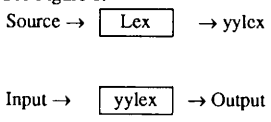
Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different com-

puter hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2]) has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.



An overview of Lex
Figure 1

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

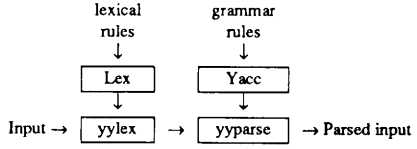
```
%%
[ \t]+$    ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.



Lex with Yacc
Figure 2

Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subrou-

tines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *cd*. . . Such backup is more costly than the processing of simpler languages.

2. Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions*, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour          printf("color");
mechandise      printf("mechandize");
petrol          printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*; a way of dealing with this will be described later.

3. Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair [ ]. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ – and ^. The – character indicates ranges. For example,

```
[a–z0–9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and under-

line. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0–z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

[–+0–9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or control characters; or

[^a–zA–Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

is the class of all characters except newline. Escaping into octal is possible although non-portable:

[\40–\176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

ab?c

matches either *ac* or *abc*.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

*a**

is any number of consecutive *a* characters, including zero; while

a+

is one or more instances of *a*. For example,

[a–z]+

is all strings of lower case letters. And

[A–Za–z][A–Za–z0–9]*

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

(ab|cd)

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary on the outside level;

ab|cd

would have sufficed. Parentheses can be used for more complex expressions:

(ab|cd+)?(ef)*

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

ab/cd

matches the string *ab*, but only if followed by *cd*. Thus

ab$

is the same as

ab/\n

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

<ONE>

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of *a*.

Finally, initial % is special, being the separator for Lex source segments.

4. Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action.

This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]    ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a–z]+*. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a–z]+    printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output.

Another way to print data is to use ECHO. ECHO stands for

```
fprintf(yyout, "%s",yytext)
```

Thus,

```
[a–z]+    ECHO;
```

is the same as

```
[a–z]+ fprintf(yyout, "%s",yytext)
```

Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a–z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a–zA–Z]+    {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng–1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless(n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \ The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*    {
           if (yytext[yyleng–1] == '\\')
              yymore();
           else
              ... normal user processing
           }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=–a". Suppose it is desired to treat this as "=– a" but print a message. A rule might be

```
=–[a–zA–Z]    {
```

```
printf("Op (=-) ambiguous\n");
yyless(yyleng-1);
... action for =- ...
}
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]    {
              printf("Op (=-) ambiguous\n");
              yyless(yyleng-2);
              ... action for = ...
              }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;
2) *output(c)* which writes the character *c* on the output; and
3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

5. Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.
2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because *[a-z]+* matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus

expressions like .* stop on the current line. Don't try to defeat this with expressions like *[.\n]+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both ***she*** and ***he*** in an input text. Some Lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides ***he*** and ***she***. Remember that . does not include newline. Since ***she*** includes ***he***, Lex will normally *not* recognize the instances of ***he*** included in ***she***, since once it has passed a ***she*** those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of ***he***:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that ***she*** includes ***he*** but not vice versa, and omit the REJECT action on ***he***; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+      { ... ; REJECT;}
a[cd]+      { ... ; REJECT;}
```

If the input is ***ab***, only the first rule matches, and on ***ad*** only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word ***the*** is considered to contain both ***th*** and ***he***. Assuming a two-dimensional array named ***digram*** to be incremented, the appropriate source is

```
%%
[a-z][a-z]      {
                digram[yytext[0]][yytext[1]]++;
                REJECT;
                }
.               ;
\n              ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6. Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only *%{* and *%}* is copied out as above. The delimiters are discarded. This format permits

entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define Lex substitution strings. The format of such lines is

name translation

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0–9]
E                       [DEde][–+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0–9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

7. Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library [6].

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same.

UNIX. The library is accessed by the loader flag *–ll*. So an appropriate set of commands is

```
lex source cc lex.yy.c –ll
```

The resulting program is placed on the usual file ***a.out*** for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of ***input***, ***output*** and ***unput*** are given, the library can be avoided.

8. Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex()*, the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex()*. In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c –ly –ll
```

The Yacc library (–ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9. Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
                int k;
[0–9]+          {
                k = atoi(yytext);
                if (k%7 == 0)
                   printf("%d", k+3);
                else
```

```
                printf("%d",k);
            }
```

to do just that. The rule [0–9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k*. The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                        int k;
-?[0–9]+                {
                        k = atoi(yytext);
                        printf("%d",
                          k%7 == 0 ? k+3 : k);
                        }
-?[0–9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if–else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
            int lengs[100];
%%
[a–z]+      lengs[yyleng]++;
.           |
\n          ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n}
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]      ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0–9]+{W}{d}{W}[+–]?{W}[0–9]+     |
[0–9]+{W}"."{W}{d}{W}[+–]?{W}[0–9]+     |
"."{W}[0–9]+{W}{d}{W}[+–]?{W}[0–9]+     {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p=+ 'e'– 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program then adds *'e'–'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}            |
{d}{c}{o}{s}            |
{d}{s}{q}{r}{t}         |
{d}{a}{t}{a}{n}         |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to

initial *a*:

```
{d}{l}{o}{g}        |
{d}{l}{o}{g}10      |
{d}{m}{i}{n}1       |
{d}{m}{a}{x}1       {
                    yytext[0] =+ 'a' – 'd';
                    ECHO;
                    }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r' – 'd';
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A–Za–z][A–Za–z0–9]*    |
[0–9]+                  |
\n                      |
                        ECHO;
```

Note that this program is not complete; it does not deal with the s,.acing problems in Fortran or with the use of keywords as identifiers.

10. Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as *$* recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
            int flag;
%%
^a          {flag = 'a'; ECHO;}
^b          {flag = 'b'; ECHO;}
^c          {flag = 'c'; ECHO;}
\n          {flag =  0 ; ECHO;}
magic       {
            switch (flag)
            {
            case 'a': printf("first"); break;
            case 'b': printf("second"); break;
            case 'c': printf("third"); break;
            default: ECHO; break;
            }
            }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start     name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                  {ECHO; BEGIN AA;}
^b                  {ECHO; BEGIN BB;}
^c                  {ECHO; BEGIN CC;}
\n                  {ECHO; BEGIN 0;}
<AA>magic           printf("first");
<BB>magic           printf("second");
<CC>magic           printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11. Character Set.

The programs generated by Lex handle character I/O only through the routines *input*, *output*, and *unput*. Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext*. For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example

```
%T
1       Aa
2       Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and – into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

12. Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form
```
%{
code
%}
```
4) Start conditions, given in the form
```
%S name1 name2 ...
```
5) Character set tables, in the form
```
%T
number space character-string
...
%T
```
6) Changes to internal array sizes, in the form
```
%x  nnn
```
where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x–z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.

x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

14. Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15. References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran*, Software – Practice and Experience, **5**, pp. 395-496 (1975).
3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Comm. ACM **18**, 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor*, Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library*, Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

M4 is a macro processor available on UNIX† and GCOS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

† UNIX is a trademark of AT&T Bell Laboratories.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is `-', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **/m4**.

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines **N** to be 100, and uses this ``symbolic constant'' in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by `(', it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains a lot of **N**'s.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both **M** and **N** to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In M4, the latter is true — **M** is 100, so even if **N** subsequently changes, **M** does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

define(M, 100)

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now **M** is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the **M** will be replaced by **N** which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ´ is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N´)
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

`define´ = 1;

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the N in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine N, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(`N', 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ´ are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N')
```

removes the definition of N. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix', `define(wordsize,16)')
ifdef(`gcos', `define(wordsize,36)')
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(`unix', on UNIX, not on UNIX)
```

Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

```
define(a,  b  c)
```

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally (b,c). And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

```
define(N, 100)
define(N1, `incr(N)')
```

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^      (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
!           (not)
& or &&     (logical and)
| or ||     (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be **2******N+1**. Then

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

```
include(filename)
```

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp:* a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

ifelse(a, b, c, d)

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns ``yes'' or ``no'' if they are the same or different.

define(compare, `ifelse($1, $2, yes, no)')

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

ifelse(a, b, c, d, e, f, g)

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

ifelse(a, b, c)

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

len(abcdef)

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

substr(`now is the time', 1)

is

ow is the time

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or −1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

translit(s, f, t)

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

translit(s, aeiou, 12345)

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

translit(s, aeiou)

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

define(N, 100)
define(M, 200)
define(L, 300)

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
    define(...)
    ...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

errprint(`fatal error')

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

```
3  changequote(L, R)
1  define(name, replacement)
4  divert(number)
4  divnum
5  dnl
5  dumpdef(`name´, `name´, ...)
5  errprint(s, s, ...)
4  eval(numeric expression)
3  ifdef(`name´, this if true, this if false)
5  ifelse(a, b, c, d)
4  include(file)
3  incr(number)
5  index(s1, s2)
5  len(string)
4  maketemp(...XXXXX...)
4  sinclude(file)
5  substr(string, position, number)
4  syscmd(s)
5  translit(str, from, to)
3  undefine(`name´)
4  undivert(number,number,...)
```

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

References

[1] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Inc., 1976.

PS19

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **termcap**(5) database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Revised 16 April 1986

Contents

Appendixes

1. Overview

In making available the generalized terminal descriptions in **termcap**(5), much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called ***terminal screen***. The package's idea of what the terminal's screen currently looks like, *i.e.*, what the user sees now. This is a special *screen*:

screen: This is a subset of windows which are as large as the terminal screen, *i.e.*, they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

#include <curses.h>

at the top of the program source. The header file **<curses.h>** needs to include **<sgtty.h>**, so the one should not do so oneself[1]. Also, compilations should have the following form:

cc [*flags*] file ... **–lcurses –ltermcap**

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *refresh()* makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window

[1] The screen package also uses the Standard I/O library, so **<curses.h>** includes **<stdio.h>**. It is redundant (but harmless) for the programmer to do it, too.

*does **not** change the terminal.* Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this", and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch()*) is provided[2]. This convention of prepending function names with a "**w**" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "**mv**" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class "register" (*e.g., reg int i;*)
bool	boolean type, actually a "char" (*e.g., bool doneit;*)
TRUE	boolean "true" flag (1).
FALSE	boolean "false" flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *initscr()* must ***always*** be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *cbreak()* should be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *delwin()* will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *addch()* adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, *i.e.*, printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, *i.e.*, that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routines *touchwin()*, *touchline()*, and *touchoverlap()* are provided to make it look like a desired part of window has been changed, thus forcing *refresh()* check that whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be cbreak, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *getttmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to initscr, *endwin()* should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as **rogue** and **vi**[3]. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "*crt hacks*"[4] and optimizing **more**(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The **termcap**(5) database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from **vi** and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, *HO* is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by **gtty**(2) and **stty**(2) The second, *setterm()*, a larger task by reading in the descriptions from the **termcap**(5) database. This is the way these routines are used by *initscr()*:

---

[3] **rogue** actually uses these functions, **vi** does not.

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as **rain**, **rocket**, and **gun**.

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

[6] These names are identical to those variables used in the **termcap**(5) database to describe each capability. See Appendix A for a complete list of those read, and the **termcap**(5) manual page for a full description.

```
if (isatty(0)) {
    gettmode();
    if ((sp=getenv("TERM")) != NULL)
        setterm(sp);
            else
                setterm(Def_term);
}
else
    setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[7]. If it is, *gettmode()* sets the terminal description modes from a **gtty**(2) *getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from **termcap**(5) associated with that terminal. (*getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def_term* is used. The *TI* and *VS* sequences initialize the terminal (*_puts()* is a macro which uses *tputs()* (see **termcap**(3)) and *_putchar()* to put out a string). *endwin()* undoes these things.

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it[8]. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor **vi** uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto()* from the **termlib**(7) routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

5. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as it's "**w**" counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

5.1. Output Functions

addch(ch) †
char ch;

[7] *isatty()* is defined in the default C library function routines. It does a **gtty**(2) on the descriptor and checks the return value.

[8] Actually, it *can* be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

waddch(win, ch)
WINDOW **win;*
char *ch;*

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline (`\n`) the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return (`\r`) will move to the beginning of the line on the window. Tabs (`\t`) will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

addstr(str) †
char **str;*

waddstr(win, str)
WINDOW **win;*
char **str;*

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

box(win, vert, hor)
WINDOW **win;*
char *vert, hor;*

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() †

wclear(win)
WINDOW **win;*

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok(scr, boolf) †
WINDOW **scr;*
bool *boolf;*

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

clrtobot() †

wclrtobot(win)
WINDOW **win;*

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "**mv**" command.

clrtoeol() †

wclrtoeol(win)
*WINDOW *win;*

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

delch()

wdelch(win)
*WINDOW *win;*

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
*WINDOW *win;*

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase() †

werase(win)
*WINDOW *win;*

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "**mv**" command.

flushok(win, boolf) †
*WINDOW *win;*
bool boolf;

Normally, *refresh() fflush()*'s *stdout* when it is finished. *flushok()* allows you to control this. if *boolf* is TRUE (*i.e.*, non-zero) it will do the *fflush()*; if it is FALSE. it will not.

idlok(win, boolf)
*WINDOW *win;*
bool boolf;

Reserved for future use. This will eventually signal to *refresh()* that it is all right to use the insert and delete line sequences when updating the window.

insch(c)
char c;

winsch(win, c)
*WINDOW *win;*
char c;

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

insertln()

winsertln(win)
*WINDOW *win;*

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged.

move(y, x) †
int y, x;

wmove(win, y, x)
*WINDOW *win;*
int y, x;

Change the current (y, x) co-ordinates of the window to (*y, x*). This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
*WINDOW *win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
*WINDOW *win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, *i.e.*, blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
*char *fmt;*

wprintw(win, fmt, arg1, arg2, ...)
*WINDOW *win;*
*char *fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll

illegally.

refresh() †

wrefresh(win)
*WINDOW *win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

As a special case, if *wrefresh()* is called with the window *curscr* the screen is cleared and repainted as it is currently. This is very useful for allowing the redrawing of the screen when the user has garbage dumped on his terminal.

standout() †

wstandout(win)
*WINDOW *win;*

standend() †

wstandend(win)
*WINDOW *win;*

Start and stop putting characters onto *win* in standout mode. *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

5.2. Input Functions

cbreak() †

nocbreak() †

crmode() †

nocrmode() †

Set or unset the terminal to/from cbreak mode. The misnamed macros *crmode()* and *nocrmode()* are retained for backwards compatibility with ealier versions of the library.

echo() †

noecho() †

Sets the terminal to echo or not echo characters.

getch() †

wgetch(win)
*WINDOW *win;*

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

getstr(str) †
char **str;*

wgetstr(win, str)
WINDOW **win;*
char **str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

_putchar(c)
char *c;*

Put out a character using the *putchar()* macro. This function is used to output every character that **curses** generates. Thus, it can be redefined by the user who wants to do non-standard things with the output. It is named with an initial "_" because it usually should be invisible to the programmer.

raw() †

noraw() †

Set or unset the terminal to/from raw mode. On version 7 UNIX[9] this also turns of newline mapping (see *nl()*).

scanw(fmt, arg1, arg2, ...)
char **fmt;*

wscanw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

baudrate() †

Returns the output baud rate of the terminal. This is a system dependent constant (defined in **<sys/tty.h>** on BSD systems, which is included by **<curses.h>**).

[9] UNIX is a trademark of Bell Laboratories.

delwin(win)
*WINDOW *win;*

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

erasechar() †

Returns the erase character for the terminal, *i.e.*, the character used by the user to erase a single character from the input.

*char **
getcap(str)
*char *str;*

Return a pointer to the **termcap** capability described by *str* (see **termcap**(5) for details).

getyx(win, y, x) †
*WINDOW *win;*
int y, x;

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

inch() †

winch(win) †
*WINDOW *win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is true, *Def_term* is always used. If the system supports the **TIOCGWINSZ** *ioctl(2)* call, it is used to get the number of lines and columns for the terminal, otherwise it is taken from the **termcap** description.

killchar() †

Returns the line kill character for the terminal, *i.e.*, the character used by the user to erase an entire line from the input.

leaveok(win, boolf) †
WINDOW ***win;**
bool *boolf;*

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
char ***termbuf, *name;**

fullname(termbuf, name)
char ***termbuf, *name;**

longname() fills in *name* with the long name of the terminal described by the **termcap** entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *termbuf* is usually set via the termlib routine *tgetent()*. *fullname()* is the same as *longname()*, except that it gives the fullest name given in the entry, which can be quite verbose.

mvwin(win, y, x)
WINDOW ***win;**
int *y, x;*

Move the home position of the window *win* from its current starting coordinates to (*y, x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything. For subwindows, *mvwin()* also returns ERR if you attempt to move it off its main window. If you move a main window, all subwindows are moved along with it.

WINDOW *
newwin(lines, cols, begin_y, begin_x)
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* − *begin_y*) or (*COLS* − *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

nl() †

nonl() †

Set or unset the terminal to/from nl mode, *i.e.*, start/stop the system from mapping **<RETURN>** to **<LINE-FEED>**. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf) †
WINDOW ***win;**
bool *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchline(win, y, startx, endx)
WINDOW ***win;**
int *y, startx, endx;*

This function performs a function similar to *touchwin()* on a single line. It marks the first change for the given line to be *startx*, if it is before the current first change mark, and the last change mark is set to be *endx* if it is currently less than *endx*.

touchoverlap(win1, win2)
WINDOW ***win1, **win2;*

Touch the window *win2* in the area which overlaps with *win1*. If they do not overlap, no changes are made.

touchwin(win)
WINDOW ***win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

WINDOW ***
subwin(win, lines, cols, begin_y, begin_x)
WINDOW ***win;**
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*) inside the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y, begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* – *begin_y*) or (*COLS* – *begin_x*) respectively.

unctrl(ch) †
char *ch;*

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their upper-case equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you may have to have **#include <unctrl.h>** in your file.

5.4. Details

gettmode()

Get the tty stats. This is normally called by *initscr()*.

mvcur(lasty, lastx, newy, newx)
int *lasty, lastx, newy, newx;*

Moves the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
WINDOW **win;*

Scroll the window upward one line. This is normally not used by the user.

savetty() †

resetty() †

savetty() saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

setterm(name)
char **name;*

Set the terminal characteristics to be those of the terminal named *name*, getting the terminal size from the **TIOCGWINSZ** *ioctl(2)* if it exists, otherwise from the environment. This is normally called by *initscr()*.

tstp()

If the new tty(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see **termcap**(5).

1.2. Overview

Capabilities from **termcap** are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, i e.g. , **12***. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

1.3. Variables Set By setterm()

variables set by *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non-Destructive space

variables set by *setterm()*

Type	Name	Pad	Description
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with ***X*** are reserved for severely nauseous glitches

For purposes of *standout()*, if *SG()* is not 0, *SO()* is set to *NULL()*, and if *UG()* is not *0()*, *US()* is set to *NULL()*. If, after this, *SO()* is *NULL()*, and *US()* is not, *SO()* is set to be *US()*, and *SE()* is set to be *UE()*.

1.4. Variables Set By gettmode()

variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

1. The WINDOW structure

The WINDOW structure is defined as follows:

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 *
 *      @(#)win_st.c        6.1 (Berkeley) 4/24/86";
 */

# define                WINDOW  struct _win_st

struct _win_st {
        short                   _cury, _curx;
        short                   _maxy, _maxx;
        short                   _begy, _begx;
        short                   _flags;
        short                   _ch_off;
        bool                    _clear;
        bool                    _leave;
        bool                    _scroll;
        char                    **_y;
        short                   *_firstch;
        short                   *_lastch;
        struct _win_st          *_nextp, *_orig;
};

# define        _ENDLINE        001
# define        _FULLWIN        002
# define        _SCROLLWIN      004
# define        _FLUSH          010
# define        _FULLLINE       020
# define        _IDLINE         040
# define        _STANDOUT       0200
# define        _NOCHANGE       -1
```

_cury and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *_maxy* and *_maxx* are the maximum values allowed for (*_cury*, *_curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, *i.e.*, the window's home. *_cury*, *_curx*, *_maxy*, and *_maxx* are measured relative to (*_begy*, *_begx*), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the *i*th line, and

[10] All variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

_y[i][j]

is the *j*th character on the *i*th line. *_flags* can have one or more values or'd into it.

For windows that are not subwindows, *_orig* is NULL . For subwindows, it points to the main window to which the window is subsidiary. *_nextp* is a pointer in a circularly linked list of all the windows which are subwindows of the same main window, plus the main window itself.

_firstch and *_lastch* are *malloc()*ed arrays which contain the index of the first and last changed characters on the line. *_ch_off* is the x offset for the window in the *_firstch* and *_lastch* arrays for this window. For main windows, this is always 0; for subwindows it is the difference between the starting point of the main window and that of the subindow, so that change markers can be set relative to the main window. This makes these markers global in scope.

All subwindows share the appropriate portions of *_y*, *_firstch*, *_lastch*, and *_insdel* with their main window.

_ENDLINE says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_FULLLINE** says that the width of a line is the same as the width of the terminal. If **_FLUSH** is set, it says that *fflush(stdout)* should be called at the end of each *refresh()* **_STANDOUT** says that all characters added to the screen are in standout mode. **_INSDEL** is reserved for future use, and is set by *idlok()*. *_firstch* is set to **_NOCHANGE** for lines on which there has been no change since the last *refresh()*.

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved. The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
static char sccsid[] = "@(#)twinkle1.c                6.1 (Berkeley) 4/24/86";
#endif not lint

# include       <curses.h>
# include       <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define        NCOLS   80
# define        NLINES  24
# define        MAXPATTERNS     4

typedef struct {
        int     y, x;
} LOCS;

LOCS    Layout[NCOLS * NLINES];         /* current board layout */

int     Pattern,                        /* current pattern number */
        Numstars;                       /* number of stars in pattern */

char    *getenv();

int     die();

main()
{
```

```
        srand(getpid());                        /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);

        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die()
{
        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS - 1, LINES - 1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard()
{
        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp->x = x;
                                lp++;
                        }
        Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
```

```
reg int         y, x; {

                switch (Pattern) {
                 case 0:                /* alternating lines */
                        return !(y & 01);
                 case 1:                /* box */
                        if (x >= LINES && y >= NCOLS)
                                return FALSE;
                        if (y < 3 || y >= NLINES - 3)
                                return TRUE;
                        return (x < 3 || x >= NCOLS - 3);
                 case 2:                /* holy pattern! */
                        return ((x + y) & 01);
                 case 3:                /* bar across center */
                        return (y >= 9 && y <= 15);
                }
                /* NOTREACHED */
}

puton(ch)
reg char                ch;
{
                reg LOCS                *lp;
                reg int                 r;
                reg LOCS                *end;
                LOCS                    temp;

                end = &Layout[Numstars];
                for (lp = Layout; lp < end; lp++) {
                        r = rand() % Numstars;
                        temp = *lp;
                        *lp = Layout[r];
                        Layout[r] = temp;
                }

                for (lp = Layout; lp < end; lp++) {
                        mvaddch(lp->y, lp->x, ch);
                        refresh();
                }
}
```

2.2. Life

This program fragment models the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This code, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved. The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
static char sccsid[] = "@(#)life.c                      6.1 (Berkeley) 4/23/86";
```

```
#endif not lint

# include       <curses.h>
# include       <signal.h>

/*
 *      Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */

typedef struct lst_st {                                 /* linked list element */
        int                     y, x;                   /* (y, x) position of piece */
        struct lst_st           *next, *last;           /* doubly linked */
} LIST;

LIST    *Head;                          /* head of linked list */

int     die();

main(ac, av)
int     ac;
char    *av[];
{
        evalargs(ac, av);                               /* evaluate arguments */

        initscr();                                      /* initialize screen package */
        signal(SIGINT, die);                            /* set to restore tty stats */
        cbreak();                                       /* set for char-by-char */
        noecho();                                       /*
        nonl();                                         /* for optimization */

        getstart();                                     /* get starting position */
        for (;;) {
                prboard();                              /* print out current board */
                update();                               /* update board position */
        }
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
die()
{
        signal(SIGINT, SIG_IGN);                        /* ignore rubouts */
        mvcur(0, COLS - 1, LINES - 1, 0);               /* go to bottom of screen */
        endwin();                                       /* set terminal to good state */
        exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
```

```
 * board setup is ready.
 */
getstart()
{
        reg char        c;
        reg int         x, y;
        auto char       buf[100];

        box(stdscr, '|', '_');                  /* box in the screen */
        move(1, 1);                             /* move to upper left corner */

        for (;;) {
                refresh();                      /* print current position */
                if ((c = getch()) == 'q')
                        break;
                switch (c) {
                 case 'u':
                 case 'i':
                 case 'o':
                 case 'j':
                 case 'l':
                 case 'm':
                 case ',':
                 case '.':
                        adjustyx(c);
                        break;
                 case 'f':
                        mvaddstr(0, 0, "File name: ");
                        getstr(buf);
                        readfile(buf);
                        break;
                 case 'x':
                        addch('X');
                        break;
                 case ' ':
                        addch(' ');
                        break;
                }
        }

        if (Head != NULL)                       /* start new list */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
}
```

```
/*
 * Print out the current board position from the linked list
 */
prboard() {

        reg LIST        *hp;

        erase();                                /* clear out last position */
        box(stdscr, '|', '_');                  /* box in the screen */

        /*
         * go through the list adding each piece to the newly
         * blank board
         */
        for (hp = Head; hp; hp = hp->next)
                mvaddch(hp->y, hp->x, 'X');

        refresh();
}
```

3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved. The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
static char sccsid[] = "@(#)twinkle2.c          6.1 (Berkeley) 4/24/86";
#endif not lint

extern int      _putchar();

main()
{
        reg char        *sp;

        srand(getpid());                        /* initialize random sequence */

        if (isatty(0)) {
            gettmode();
            if ((sp = getenv("TERM")) != NULL)
                    setterm(sp);
            signal(SIGINT, die);
        }
        else {
                printf("Need a terminal on %d\n", _tty_ch);
                exit(1);
```

```
        }
        _puts(TI);
        _puts(VS);

        noecho();
        nonl();
        tputs(CL, NLINES, _putchar);
        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

puton(ch)
char    ch;
{
        reg LOCS        *lp;
        reg int         r;
        reg LOCS        *end;
        LOCS            temp;
        static int      lasty, lastx;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++)
                        /* prevent scrolling */
                if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                        mvcur(lasty, lastx, lp->y, lp->x);
                        putchar(ch);
                        lasty = lp->y;
                        if ((lastx = lp->x + 1) >= NCOLS)
                                if (AM) {
                                        lastx = 0;
                                        lasty++;
                                }
                                else
                                        lastx = NCOLS - 1;
                }
}
```

Appendix A
make

Appendix A: make

Introduction

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

make(1) provides a method for maintaining up-to-date versions of programs that consist of a number of files that may be generated in a variety of ways.

An individual programmer can easily forget

- file-to-file dependencies
- files that were modified and the impact that has on other files
- the exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of **make** is to

- find the target in the description file
- ensure that all the files on which the target depends, the files needed to generate the target, exist and are up to date
- create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called **makefile**, **Makefile**, or **s.[mM]akefile**. If this naming convention is followed, the simple command **make** is usually sufficient to regenerate the target regardless of the number files edited since the last **make**. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than typing all the commands to regenerate the target, typing the **make** command ensures the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up to date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program searches the graph of dependencies. The operation of **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- a user-supplied description file
- filenames and last-modified times from the file system
- built-in rules to bridge some of the gaps

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c**, and **z.c** with the **math** library. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o**, and **z.o**. Assume that the files **x.c** and **y.c** share some declarations in a file named **defs.h**, but that **z.c** does not. That is, **x.c** and **y.c** have the line

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o -o  prog

x.o  y.o :   defs.h
```

If this information were stored in a file named **makefile**, the command

```
make
```

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c**, or **defs.h**. In the example above, the first line states that **prog** depends on three **.o** files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that **x.o** and **y.o** depend on the file **defs.h**. From the file system, **make** discovers that there are three **.c** files corresponding to the needed **.o** files and uses built-in rules on how to generate an object from a C source file (i.e., issue a **cc –c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -o  prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, the command **make** announces this fact and stops. If, however, the **defs.h** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled; and then **prog** is created from the new **x.o** and **y.o** files, and the existing **z.o** file. If only the file **y.c** had changed, only it is recompiled; but it is still necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

make x.o

would regenerate **x.o** if **x.c** or **defs.h** had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros (for information about macros, see "Description Files and Substitutions" below.) Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean" might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equals sign followed by what the macro stands for. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be

parenthesized. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

$***, **$@, **$?**, **$<**, and **$%** are five special macros that change values during the execution of the command. (These five macros are described later in this chapter under "Description Files and Substitutions.") The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -ll
prog: $(OBJECTS)
        cc $(OBJECTS)   $(LIBES)   -o prog
  . . .
```

The command

make LIBES="-ll -ly"

loads the three objects with both the **lex** (**-ll**) and the **yacc** (**-ly**) libraries, because macro definitions on the command line override definitions in the description file. (In UNIX system commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -ll -ly
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make:  $(OBJECTS)
        $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        @size make

$(OBJECTS): defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

lint :  dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c \
        gram.c

                # print files that are out-of-date
                # with respect to "print" file.

print:   $(FILES)
         pr $? | $(LP)
         touch print
```

The **make** program prints out each command before issuing it.

The following output results from typing the command **make** in a directory containing only the source and description files:

```
cc  -O -c main.c
cc  -O -c doname.c
cc  -O -c misc.c
cc  -O -c files.c
cc  -O -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -O -c gram.c
cc  main.o doname.o misc.o files.o dosys.o
     gram.o  -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an at sign, @, in the description file.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is that a pound, #, and all characters on the same line after a pound are ignored. Blank lines and lines beginning with a pound are totally ignored.

Continuation Lines

If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

Macro Definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lcurses
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules. (See Figure 13-2 at the end of the chapter.)

General Form

The general form of an entry in a description file is

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
 . . .
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as * and ? are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a pound, #, except when the pound is in quotes.

Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the "Archive Libraries" section later in this chapter.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (–s option of the **make** command) or if the command line in the description file begins with an @ sign. **make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the –i flag has been specified on the **make** command line, if

the fake target name .IGNORE appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The **$?** macro is set to the string of names that were found to be younger than the target. The **$?** macro is evaluated when explicit rules from the **makefile** are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; and the **$*** macro is the prefix shared by the current and the dependent filenames. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name DEFAULT are used. If there is no such name, **make** prints a message and stops. The **$%** macro is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parenthesis.

In addition, a description file may also use the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)** (see below).

Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single character macro. The **F** refers to the filename part of the single character macro. These additions are useful when building hierarchical **makefiles**. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a command can be

```
cd $(<D); $(MAKE) $(<F)
```

Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of **string1** in the evaluated macro, **string2** is substituted. The meaning of finding **string1** in **$(macro)** is that the evaluated **$(macro)** is considered as a series of strings each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script, which can handle all the C language programs (i.e., those files ending in **.c**). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

Recursive Makefiles

Another feature of **make** concerns the environment and recursive invocations. If the sequence $(MAKE) appears anywhere in a shell command line or the line begins a + sign, the line is executed even if the **–n** flag is set. Since the **–n** flag is exported across invocations of **make** (through the MAKEFLAGS or MFLAGS variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of **makefile**(s) describes a set of software subsystems. For testing purposes, **make –n ...** can be executed and everything that would have been done will be printed including output from lower level invocations of **make**.

Suffixes and Transformation Rules

make reads its rule information from **/usr/lib/make/local**, or if this file does not exist then **/usr/lib/make/standard**. It uses this information to learn how to transform a file with one suffix into a file with another suffix. If the **–r** flag is used on the **make** command line, the rules file will not be read.

The list of suffixes is actually the dependency list for the name .SUFFIXES. **make** searches for a file with any of the suffixes on the list. If it finds one, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a **.r** file to a **.o** file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **$*** is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro **$<** is the full name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for .SUFFIXES in the description file. The dependents are added to the usual list. A .SUFFIXES line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	Object file
.bin	Apollo object file
.c[~]	C source file
.a	Archive file
.f[~]	FORTRAN source file
.y[~]	**yacc** source grammar
.l[~]	**lex** source grammar
.h[~]	Header file
.sh[~]	Shell file
.sh[~]	C-Shell file
.cxx[~]	C++ source file
.hxx[~]	C++ header file
.pas[~]	Apollo PASCAL source file
.ftn[~]	Apollo FORTRAN source file
.asm[~]	Apollo Assembler source file
.F[~]	FORTRAN source file
.rat[~]	RATFOR source file

Figure 13-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

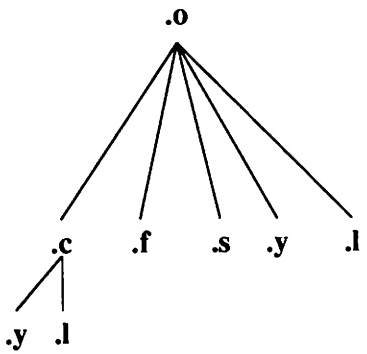


Figure A-1: Summary of Default Transformation Path

If the file **x.o** is needed and an **x.c** is found in the description or directory, the **x.o** file would be compiled. If there is also an **x.l**, that source file would be run through **lex** before compiling the result. However, if there is no **x.c** but there is an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Figure 13-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler macro names are ASM, CC, F77, YACC, LEX, CCXX, FTN, PAS, RF, ACC, and ACCXX. The command

make CC=newcc

will cause the **newcc** command to be used instead of the usual C language compiler. The macros ASMFLAGS, CFLAGS, F77FLAGS, YFLAGS, LFLAGS, CCXXFLAGS, FFLAGS, PFLAGS, RFLAGS, ACFLAGS, and ACCXXFLAGS may be set to cause these commands to be issued with optional flags. Thus

make "CFLAGS=-g"

causes the **cc** command to include debugging information.

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library in the following manner:

```
projlib(object.o)
   or
projlib((entrypt))
```

where the second method actually refers to an entry point of an object file within the library. (**make** looks through the library, locates the entry point, and translates it to the correct object filename.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib::   projlib(pfile1.o)
        $(CC) -c -O pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        $(RM) pfile1.o
projlib::   projlib(pfile2.o)
        $(CC) -c -O pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        $(RM) pfile2.o
```

... and so on for each object ...

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the filename being the only difference each time. (This is true in most cases.)

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.asm.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.asm~.a**. (The tilde, ~, syntax will be described shortly.) The user may define other needed rules in the description file.

The above two-member library is then maintained with the following shorter **makefile**:

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
              @echo projlib up-to-date
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:

        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o
```

Thus, the **$@** macro is the **.a** target (**projlib**); the **$<** and **$*** macros are set to the out-of-date C language file, and the filename minus the suffix, respectively (**pfile1.c** and **pfile1**). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction

```
projlib:    projlib(pfile1.o)
          @echo projlib up-to-date
```

Assume the object in the library is out of date with respect to **pfile1.c**. Also, there is no **pfile1.o** file.

1. **make projlib**.
2. Before **make**ing **projlib**, check each dependent of **projlib**.
3. **projlib(pfile1.o)** is a dependent of **projlib** and needs to be generated.
4. Before generating **projlib(pfile1.o)**, check each dependent of **projlib(pfile1.o)**. (There are none.)
5. Use internal rules to try to create **projlib(pfile1.o)**. (There is no explicit rule.) Note that **projlib(pfile1.o)** has a parenthesis in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parenthesis implies the **.a** suffix. In this sense, the **.a** is hard-wired into **make**.
6. Break the name **projlib(pfile1.o)** up into **projlib** and **pfile1.o**. Define two macros, **$@** (=**projlib**) and **$*** (=**pfile1**).

7. Look for a rule *.X*.**a** and a file **$***.*X*. The first *.X* (in the .SUFFIXES list) which fulfills these conditions is **.c** so the rule is **.c.a**, and the file is **pfile1.c**. Set **$<** to be **pfile1.c** and execute the rule. In fact, **make** must then compile **pfile1.c**.

8. The library has been updated. Execute the command associated with the **projlib:** dependency, namely

```
@echo projlib up-to-date
```

It should be noted that to let **pfile1.o** have dependencies, the following syntax is required:

```
projlib(pfile1.o):        $(INCDIR)/stdio.h  pfile1.c
```

The **$%** macro can be used for referencing the archive member name when this form is used.

Source Code Control System Filenames: the Tilde

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, **s.** precedes the filename part of the complete path name.

To allow **make** easy access to the prefix **s.** the tilde, ~, is used as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the default rule is

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -$(RM) -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS filename search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

SCCSDIR is another special macro. The SCCSDIR macro, when specified as SCCS, allows **make** to find **s.** files in the sub-directory **SCCS**. Otherwise, **make** will onlyu find **s.** files in the current directory.

The following SCCS suffixes are defined by default:

.c~ .h~ .cxx~ .hxx~ .pas~ .f~ .r~ .F~ .ftn~ .rat~ .y~ .l~ .asm~ .sh~ .csh~

The following rules involving SCCS transformations are defined by default:

.c~: .c~.o: .c~.a: .c~.bin: .c~.c: .h~.h: .cxx~: .cxx~.o: .cxx~.a: .cxx~.bin: .cxx~.cxx: .hxx~.hxx: .pas~: .pas~.o: .pas~.a: .pas~.bin: .f~: .r~: .F~: .f~.o: .r~.o: .F~.o: .f~.a: .r~.a: .F~.a: .ftn~: .ftn~.o: .ftn~.a: .ftn~.bin: .rat~: .rat~.ftn: .rat~.o: .rat~.a: .rat~.bin: .y~: .y~.c : .y~.o: .y~.bin: .l~: .l~.c: .l~.o: .l~.bin: .asm~: .asm~.o: .asm~.a: .asm~.bin: .sh~: .sh~.sh: .csh~: .csh~.csh:

Obviously, the user can define other rules and suffixes, which may prove useful. The tilde provides a handle on the SCCS filename format so that this is possible.

The Null Suffix

There are many programs that consist of a single source file. **make** handles this case by the null suffix rule. Thus, to maintain the UNIX system program **cat**, a rule in the **makefile** of the following form is needed:

```
.c:
        $(CC) $(CFLAGS) $< -o $@
```

In fact, this **.c:** rule is internally defined so no **makefile** is necessary at all. The user only needs to type

make cat dd echo date

(these are all UNIX system single-file programs) and all four C language source files are passed through the above shell command line associated with the **.c:** rule. The single suffix rules defined by default are

.c: .cxx: .pas: .f: .r: .F: .f~: .r~: .F~: .ftn: .ftn~: .rat: .rat~: .y: .y~: .l: .l~: .asm: .asm~: .sh: .sh~: .csh: .csh~:

Others may be added in the **makefile** by the user.

include Files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the rest of the line is assumed to be a filename, which the current invocation of **make** will read. Macros may be used in filenames. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **include**s are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named **s.makefile** or **s.Makefile** or **$(SCCSDIR)/s.makefile** or **$(SCCSDIR)/s.Makefile** exists, **make** will do a **get** on the file, then read and remove the file.

Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The **$$@** refers to the current "thing" to the left of the colon (which is **$@**). Also the form **$$(@F)** exists, which allows access to the file part of **$@**. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX software command directory could have a **makefile** like:

```
CMDS = cat dd echo date cmp comm chown

$(CMDS):        $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile.**

The second useful form of the dependency parameter is **$$(@F)**. It represents the filename part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus, the **/usr/src/head/makefile** would look like

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

Command Usage

The **make** command description is found under **make**(1) in the *Programmer's Reference Manual.*

The make Command

The **make** command takes macro definitions, options, description filenames, and target filenames as arguments in the form:

make [*options*] [*macro definitions*] [*targets*]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ sign are printed.

–t Touch the target files (causing them to be up to date) rather than issue the usual commands.

–q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.

–p Print out the complete set of macro definitions and target descriptions.

–k Abandon work on the current entry if something goes wrong, but continue on other branches that do not depend on the current entry.

–e Environment variables override assignments within **makefiles**.

–f Description filename. The next argument is assumed to be the name of a description file. A filename of – denotes the standard input. If there are no **–f** arguments, the file named **makefile** or **Makefile** or **s.[mM]akefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.

The following two arguments are evaluated in the same manner as flags:

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.

.PRECIOUS Dependents on this target are not removed when quit or interrupt is pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A macro, MAKEFLAGS, is maintained by **make**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the **makefile** update MAKEFLAGS. Thus, to describe how the environment interacts with **make**, the MAKEFLAGS macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal **make** variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the **–f**, **–p**, and **–r** flags.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).

4. Read the **makefile**(s). The assignments in the **makefile**(s) override the environment. This order is chosen so that when a **makefile** is read and executed, you know what to expect. That is, you get what is seen unless the **–e** flag is used. The **–e** is the line flag, which tells **make** to have the environment override the **makefile** assignments. Thus, if **make –e ...** is typed, the variables in the environment override the definitions in the **makefile**. Also MAKEFLAGS override the environment if assigned. This is useful for further invocations of **make** from the current **makefile**.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions
2. environment
3. **makefile**(s)
4. command line

The **–e** flag has the effect of rearranging the order to:

1. internal definitions
2. **makefile**(s)
3. environment
4. command line

This order is general enough to allow a programmer to define a **makefile** or set of **makefiles** whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file **x.c** has a

```
#include "defs.h"
```

line, then the object file **x.o** depends on **defs.h**; the source file **x.c** does not. If **defs.h** is changed, nothing is done to the file **x.c** while file **x.o** must be recreated.

To discover what **make** would do, the **–n** option is very useful. The command

make –n

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a comment to an **include** file), the **–t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

make –ts

(touch silently) causes the relevant files to appear up to date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Default Rules

The standard set of rules used by **make** are reproduced below. These rules can be found in **/usr/lib/make/standard**.

```
#
#       Suffixes recognized by make
#
        .SUFFIXES: .o .c .h .c~ .h~ .bin .a .cxx .hxx .cxx~
.hxx~
        .SUFFIXES: .pas .pas~ .bin .f .r .F .f~ .r~ .F~ .ftn
.ftn~
        .SUFFIXES: .rat .rat~ .y .y~ .l .l~ .asm .asm~ .sh
.sh~ .csh .csh~
#
#       Predefined macros
#
ACC=/com/cc                     FFLAGS=-opt
ACCXX=/com/ccxx                 FTN=/com/ftn
ACCXXFLAGS=-opt                 GET=get
ACFLAGS=-opt                    GFLAGS=
AR=ar                           LD=ld
ARFLAGS=rv                      LDFLAGS=
ASM=asm                         LEX=lex
ASMFLAGS=-nl                    LFLAGS=
BIND=/com/bind                  MAKE=make
BINDFLAGS=                      MV=mv
CC=/bin/cc                      PAS=/com/pas
CCXX=ccxx                       PFLAGS=-opt
CCXXFLAGS=-O                    RF=ratfor
CFLAGS=-O                       RFLAGS=
CHMOD=chmod                     RM=rm
CP=cp                           YACC=yacc
F77=f77                         YFLAGS=
F77FLAGS=
```

make Default Rules (Sheet 1 of 12)

```
#
# Rules for C compiler.
#
.c:
        $(CC) $(CFLAGS) $< -o $@ $(LDFLAGS)
.c.o:
        $(CC) $(CFLAGS) -c $<
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o
.c.bin:
        $(ACC) $< $(ACFLAGS)
.c~:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) $*.c -o $* $(LDFLAGS)
        -$(RM) -f $*.c
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -$(RM) -f $*.c
.c~.a:
        $(GET) $(GFLAGS) $<
        $(CC) -c $(CFLAGS) $*.c
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.[co]
.c~.bin:
        $(GET) $(GFLAGS) $<
        $(ACC) $*.c $(ACFLAGS)
        -$(RM) -f $*.c
.c~.c:
        $(GET) $(GFLAGS) $<
.h~.h:
        $(GET) $(GFLAGS) $<
```

make Default Rules (Sheet 2 of 12)

```
#
# Rules for C++ compiler.
#
.cxx:
        $(CCXX) $(CCXXFLAGS) $< -o $@ $(LDFLAGS)
.cxx.o:
        $(CCXX) $(CCXXFLAGS) -c $<
.cxx.a:
        $(CCXX) -c $(CCXXFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o
.cxx.bin:
        $(ACCXX) $< $(ACCXXFLAGS)
.cxx~:
        $(GET) $(GFLAGS) $<
        $(CCXX) $(CCXXFLAGS) $*.cxx -o $* $(LDFLAGS)
        -$(RM) -f $*.cxx
.cxx~.o:
        $(GET) $(GFLAGS) $<
        $(CCXX) $(CCXXFLAGS) -c $*.cxx
        -$(RM) -f $*.cxx
.cxx~.a:
        $(GET) $(GFLAGS) $<
        $(CCXX) -c $(CCXXFLAGS) $*.cxx
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.cxx $*.o
.cxx~.bin:
        $(GET) $(GFLAGS) $<
        $(ACCXX) $*.cxx $(ACCXXFLAGS)
        -$(RM) -f $*.cxx
.cxx~.cxx:
        $(GET) $(GFLAGS) $<
.hxx~.hxx:
        $(GET) $(GFLAGS) $<
```

make Default Rules (Sheet 3 of 12)

```
#
# Rules for Apollo pascal compiler.
#
.pas:
        $(PAS) $< $(PFLAGS)
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.bin
.pas.o:
        $(PAS) $< $(PFLAGS)
        $(MV) $*.bin $@
.pas.a:
        $(PAS) $< $(PFLAGS)
        $(MV) $*.bin $*.o
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o
.pas.bin:
        $(PAS) $< $(PFLAGS)
.pas~:
        $(GET) $(GFLAGS) $<
        $(PAS) $*.pas $(PFLAGS)
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.pas $*.bin
.pas~.o:
        $(GET) $(GFLAGS) $<
        $(PAS) $*.pas $(PFLAGS)
        $(MV) $*.bin $@
        -$(RM) -f $*.pas
.pas~.a:
        $(GET) $(GFLAGS) $<
        $(PAS) $*.pas $(PFLAGS)
        $(MV) $*.bin $*.o
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o $*.pas
.pas~.bin:
        $(GET) $(GFLAGS) $<
        $(PAS) $*.pas $(PFLAGS)
        -$(RM) $*.pas
```

make Default Rules (Sheet 4 of 12)

```
#
# Rules for fortran 77 compiler.
#
	$(F77) $(F77FLAGS) $< -o $@ $(LDFLAGS)
	$(F77) $(F77FLAGS) $(RFLAGS) $(FFLAGS) -c $<
	$(F77) $(F77FLAGS) -c $<
	$(AR) $(ARFLAGS) $@ $*.o
	-$(RM) -f $*.o
.f~:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) $*.f -o $* $(LDFLAGS)
	-$(RM) -f $*.f
.r~:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) $*.r -o $* $(LDFLAGS)
	-$(RM) -f $*.r
.F~:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) $*.F -o $* $(LDFLAGS)
	-$(RM) -f $*.F
.f~.o:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) -c $*.f
	-$(RM) -f $*.f
.r~.o:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) -c $*.r
	-$(RM) -f $*.r
.F~.o:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) -c $*.F
	-$(RM) -f $*.F
.f~.a:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) -c $*.f
	$(AR) $(ARFLAGS) $@ $*.o
	-$(RM) -f $*.[fo]
.r~.a:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) -c $*.r
	$(AR) $(ARFLAGS) $@ $*.o
	-$(RM) -f $*.[ro]
.F~.a:
	$(GET) $(GFLAGS) $<
	$(F77) $(F77FLAGS) -c $*.F
	$(AR) $(ARFLAGS) $@ $*.o
	-$(RM) -f $*.[Fo]
```

make Default Rules (Sheet 5 of 12)

```
#
# Rules for Apollo fortran compiler.
#
.ftn:
        $(FTN) $< $(FFLAGS)
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.bin
.ftn.o:
        $(FTN) $< $(FFLAGS)
        $(MV) $*.bin $@
.ftn.a:
        $(FTN) $< $(FFLAGS)
        $(MV) $*.bin $@
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o
.ftn.bin:
        $(FTN) $< $(FFLAGS)
.ftn~:
        $(GET) $(GFLAGS) $<
        $(FTN) $*.ftn $(FFLAGS)
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.bin *.ftn
.ftn~.o:
        $(GET) $(GFLAGS) $<
        $(FTN) $*.ftn $(FFLAGS)
        $(MV) $*.bin $@
        -$(RM) -f $*.ftn
.ftn~.a:
        $(GET) $(GFLAGS) $<
        $(FTN) $*.ftn $(FFLAGS)
        $(MV) $*.bin $@
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.ftn $*.o
.ftn~.bin:
        $(GET) $(GFLAGS) $<
        $(FTN) $*.ftn $(FFLAGS)
        -$(RM) -f $*.ftn
```

make Default Rules (Sheet 6 of 12)

```
#
# Rules for ratfor translator.
#
.rat:
        $(RF) $< $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.ftn $*.bin
.rat.ftn:
        $(RF) $< $(RFLAGS) >$*.ftn
.rat.o:
        $(RF) $< $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        $(MV) $*.bin $@
        -$(RM) -f $*.ftn
.rat.a:
        $(RF) $< $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        $(MV) $*.bin $@
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o $*.ftn
.rat.bin:
        $(RF) $< $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        -$(RM) -f $*.ftn
.rat~:
        $(GET) $(GFLAGS) $<
        $(RF) $*.rat $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.rat $*.ftn $*.bin
.rat~.ftn:
        $(GET) $(GFLAGS) $<
        $(RF) $*.rat $(RFLAGS) >$*.ftn
        -$(RM) -f $*.rat
.rat~.o:
        $(GET) $(GFLAGS) $<
        $(RF) $*.rat $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        $(MV) $*.bin $@
        -$(RM) -f $*.rat $*.ftn
.rat~.a:
        $(GET) $(GFLAGS) $<
        $(RF) $*.rat $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        $(MV) $*.bin $@
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o $*.ftn $*.rat
.rat~.bin:
        $(GET) $(GFLAGS) $<
        $(RF) $*.rat $(RFLAGS) > $*.ftn
        $(FTN) $*.ftn
        -$(RM) -f $*.ftn $*.rat
```

make Default Rules (Sheet 7 of 12)

```
#
# Rules for YACC.
#
.y:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c -o $@ $(LDFLAGS)
        -$(RM) -f y.tab.c
        $(YACC) $(YFLAGS) $<
        $(MV) y.tab.c $@
.y.o:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        $(MV) y.tab.o $@
        -$(RM) -f y.tab.c
.y.bin:
        $(YACC) $(YFLAGS) $<
        $(ACC) y.tab.c $(ACFLAGS)
        $(MV) y.tab.bin $@
        -$(RM) -f y.tab.c
.y~:
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        $(CC) $(CFLAGS) -c y.tab.c -o $@ $(LDFLAGS)
        -$(RM) -f y.tab.c $*.y
.y~.c :
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        $(MV) y.tab.c $*.c
        -$(RM) -f $*.y
.y~.o:
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        $(CC) $(CFLAGS) -c y.tab.c
        $(MV) y.tab.o $*.o
        -$(RM) -f y.tab.c $*.y
.y~.bin:
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        $(ACC) y.tab.c $(ACFLAGS)
        $(MV) y.tab.bin $@
        -$(RM) -f y.tab.c $*.y
```

make Default Rules (Sheet 8 of 12)

```
#
# Rules for Lex.
#
.l:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c -o $@ $(LDFLAGS)
	$(RM) -f lex.yy.c
.l.c:
	$(LEX) $(LFLAGS) $<
	$(MV) lex.yy.c $@
.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	$(MV) lex.yy.o $@
	-$(RM) -f lex.yy.c
.l.bin:
	$(LEX) $(LFLAGS) $<
	$(ACC) lex.yy.c $(ACFLAGS)
	$(MV) lex.yy.bin $@
	-$(RM) -f lex.yy.c
.l~:
	$(GET) $(GFLAGS) $<
	$(LEX) $(LFLAGS) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c -o $@ $(LDFLAGS)
	-$(RM) -f lex.yy.c $*.l
.l~.c:
	$(GET) $(GFLAGS) $<
	$(LEX) $(LFLAGS) $*.l
	$(MV) lex.yy.c $@
	-$(RM) -f $*.l
.l~.o:
	$(GET) $(GFLAGS) $<
	$(LEX) $(LFLAGS) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c
	$(MV) lex.yy.o $*.o
	-$(RM) -f lex.yy.c $*.l
.l~.bin:
	$(GET) $(GFLAGS) $<
	$(LEX) $(LFLAGS) $*.l
	$(ACC) lex.yy.c $(ACFLAGS)
	$(MV) lex.yy.bin $@
	-$(RM) -f lex.yy.c $*.l
```

make Default Rules (Sheet 9 of 12)

```
#
# Rules for Apollo Assembler.
#
.asm:
        $(ASM) $< $(ASMFLAGS)
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.bin
.asm.o:
        $(ASM) $< $(ASMFLAGS)
        $(MV) $*.bin $@
.asm.a:
        $(ASM) $< $(ASMFLAGS)
        $(MV) $*.bin $*.o
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o
.asm.bin:
        $(ASM) $< $(ASMFLAGS)
.asm~:
        $(GET) $(GFLAGS) $<
        $(ASM) $*.asm $(ASMFLAGS)
        $(BIND) $*.bin -b $@ $(BINDFLAGS)
        -$(RM) -f $*.bin $*.asm
.asm~.o:
        $(GET) $(GFLAGS) $<
        $(ASM) $*.asm $(ASMFLAGS)
        $(MV) $*.bin $@
        -$(RM) -f $*.asm
.asm~.a:
        $(GET) $(GFLAGS) $<
        $(ASM) $*.asm $(ASMFLAGS)
        $(MV) $*.bin $*.o
        $(AR) $(ARFLAGS) $@ $*.o
        -$(RM) -f $*.o $*.asm
.asm~.bin:
        $(GET) $(GFLAGS) $<
        $(ASM) $*.asm $(ASMFLAGS)
        -$(RM) -f $*.asm
```

make Default Rules (Sheet 10 of 12)

```
#
# Rules for Bourne shells.
#
.sh:
        $(CP) $< $@; $(CHMOD) 0777 $@
.sh~:
        $(GET) $(GFLAGS) $<
        $(CP) $*.sh $*; $(CHMOD) 0777 $@
        -$(RM) -f $*.sh
.sh~.sh:
        $(GET) $(GFLAGS) $<
```

make Default Rules (Sheet 11 of 12)

```
#
# Rules for c-shells scripts.
#
.csh:
        $(CP) $< $@; $(CHMOD) 0777 $@
.csh~:
        $(GET) $(GFLAGS) $<
        $(CP) $*.csh $*; $(CHMOD) 0777 $@
        -$(RM) -f $*.csh
.csh~.csh:
        $(GET) $(GFLAGS) $<
```

make Default Rules (Sheet 12 of 12)

Appendix B
mk

Appendix B: mk

mk: a successor to make

mk is an efficient general tool for describing and maintaining dependencies between files or programs. **mk** is styled on the UNIX System tool **make**. The major advantages of **mk** over **make** are executing recipes in parallel, using pattern-matching metarules rather than suffix transformation rules, and deriving dependencies by transitive closure on all rules. **mk** runs anywhere from 2 to 30 times faster than **make**.

This report describes release 2 (August 1988) of **mk** mainly by means of an evolving example. It also summarizes the differences between **mk** and **make** and discuss the principles underlying **mk**'s design.

Introduction

A large fraction of computer activity consists of repeated application of tools (special or general purpose programs) to input files to produce output files. The most obvious example is programming, but other no less important examples range from simple document-processing pipelines to the generation of a circuit board or integrated circuit involving hundreds of files. Common to all these activities are file dependencies, where changing a file requires that other files be remade. **mk** reads a dependency description (called a **mk**file) and does the minimal work necessary to bring a target file up to date.

mk owes much to **make**, written by Stu Feldman, which has been doing a similar job on UNIX systems since 1976. The version of **make** referred to throughout this report is Feldman's research version distributed with the Research UNIX System, Eighth Edition and is substantially more advanced than the versions found in System V or BSD.

The next section is rather long. It follows the gradual development of a somewhat complicated mkfile describing how to build a C program. It is followed by a section on fancy uses of **mk** . The fourth section summarizes the differences between **mk** and **make** and includes a comparison of execution times. The fifth section highlights the principles underlying **mk** . The first appendix documents the predefined or builtin variables and rules for **mk** . The second appendix describes the changes made for the second release.

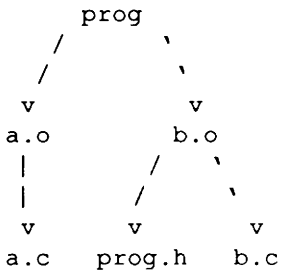
An Extended Example

This section describes **mk** in the context of building C programs. This is for the reader's comfort; **mk** knows nothing special about C programs (other than the rules in Appendix 1). The example starts off small and simple and is extended throughout the section. Sometimes, **mk** 's behavior is best demonstrated by excerpts from a terminal session. These will be shown as

```
$ date
Fri Feb 20 20:06:03 EST 1987
$
```

where $ is the prompt for the next command. Comments within examples will be shown in underlined text.

Initially, our program is called *prog* and is made from *a.o* and *b.o* , which are made by compiling *a.c* and *b.c* respectively. In addition, *b.c* includes a header file *prog.h* . We represent these relationships pictorially below



The line (arrow) means "depends on." Thus, *prog* depends on *a.o* and *b.o* and if *a.o* or *b.o* is modified, then *prog* needs to be rebuilt. Similarly, *a.o* depends on *a.c* and *b.o* depends on *b.c* and *prog.h* .

The textual description of how *prog* is built is kept in a **mk**file and looks like

```
prog:  a.o b.o
       cc -o prog a.o b.o
a.o:   a.c
       cc -c a.c
b.o:   b.c prog.h
       cc -c b.c
```

The mkfile is a sequence of *rules* . Each rule defines a target (say *prog*) that depends on some prerequisites (*a.o* and *b.o*) and the commands (a shell script called the *recipe*) to

bring the target up to date. **mk** takes this description from a file named **mk**file and builds the given targets. If no targets are given on the command line, the first target in the mkfile is built. For example, if we start with just the source files in our directory, **mk** creates *prog* by compiling *a.c* and *b.c* .

```
$ mk
cc -c a.c
cc -c b.c
cc -o prog a.o b.o
$
```

Executing **mk** again does nothing, as *prog* is now up to date.

```
$ mk
mk: 'prog' is up to date
$
```

If we change a source file, **mk** rebuilds only the files that are out of date:

```
modify a.c
$ mk
cc -c a.c
cc -o prog a.o b.o
$
```

mk will explain why it is rebuilding a file if we use the *-e* option. For example,

```
modify prog.h
$ mk -e
b.o(540869437) < prog.h(540869535)
cc -c b.c
prog(540869493) < b.o(540869546)
cc -o prog a.o b.o
$
```

Thus, *b.o* was out of date with respect to *prog.h* . After *b.o* was remade, *prog* was found to be out of date with respect to *b.o* and was then rebuilt. The numbers are the actual time stamps of the files: the values are not as important as the difference between them. A time stamp of zero indicates a non-existent file.

Variables

Suppose we now need to compile the source files with the *-g* flag so that we can use the debugger. We can of course simply edit each rule to change *cc* into *"cc -g"* :

```
prog:  a.o b.o
       cc -g -o prog a.o b.o
a.o:   a.c
       cc -g -c a.c
b.o:   b.c prog.h
       cc -g -c b.c
```

A better solution is to use a *variable* . A **mk** variable has a similar form and use to a shell variable. A suitable (mnemonic) name is *CFLAGS* . The new mkfile looks like this:

```
CFLAGS=-g
prog:  a.o b.o
       cc $CFLAGS -o prog a.o b.o
a.o:   a.c
       cc $CFLAGS -c a.c
b.o:   b.c prog.h
       cc $CFLAGS -c b.c
```

Now, if we want to profile *prog* (which means compiling everything with the *-p* option), we need only change the first line to

```
CFLAGS=-g -p
```

and recompile all the object files. The easiest way to recompile everything is with *"mk -a"* which says to always make every target regardless of time stamps.

Some variables are supplied by **mk** for use by the recipe. One is *prereq* whose value is all the prerequisites for this rule. We can rewrite the first rule like this:

```
prog:  a.o b.o
       cc $CFLAGS -o prog $prereq
```

This guarantees that the lists of object files (the prerequisite line and the cc line) are the same. It is now easy to incorporate a new object file *c.o* by adding the new name just once:

```
CFLAGS=-g -p
prog:  a.o b.o c.o
       cc $CFLAGS -o prog $prereq
a.o:   a.c
       cc $CFLAGS -c a.c
b.o:   b.c prog.h
       cc $CFLAGS -c b.c
c.o:   c.c prog.h
       cc $CFLAGS -c c.c
```

Variables get their initial value by the following rules (in decreasing order of precedence):

[1] command line assignments

[2] assignments within the mkfile

[3] values inherited from the environment

[4] default values as defined in Appendix 1.

After a variable gets its initial value, subsequent values can only be assigned in the mkfile.

```
$ cat mkfile
SYSTEM=-DV9
CFLAGS=-g
CFLAGS="$CFLAGS $SYSTEM"
printcflags:Q:
       echo $CFLAGS
$ mk
-g -DV9
$ mk SYSTEM=-DSYSTEMV
-g -DSYSTEMV
$ mk CFLAGS=-O
-O -DV9
```

Metarules

The rules for all the *.o* files are very similar. **mk** supports *metarules* , that is, rules that apply to a class of targets, rather than just one specific target. The class of targets is defined by pattern matching, with the symbol % (called the stem) equivalent to the regular expression .* . For example, the normal rule for compiling C source files is

```
%.o:    %.c
        $CC $CFLAGS -c $stem.c
```

The variable *stem* in the recipe is the string matched by the % . The % can appear anywhere in the target or prerequisite, not just at the beginning. The *CC* variable is good planning; a different compiler can be used very easily. Using this metarule, our mkfile becomes shorter:

```
CC=cc
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o:    prog.h
c.o:    prog.h
%.o:    %.c
        $CC $CFLAGS -c $stem.c
```

Notice that the prerequisites for a target can be spread across many rules. Two rules apply to *b.o* , the specific rule with *prog.h* and the metarule for *.o 's*. Metarules with recipes do not match targets of non-metarules with recipes. Thus, metarules for generating *.o 's* (say) do not conflict with any rule for generating a specific *.o* . Only one of the rules for a target should have a recipe. If there is more than one recipe, **mk** complains that the way to make the target is ambiguous.

mk has some predefined variables and rules listed in Appendix 1. Because our rule for *%.o* and the value for *CC* are the same as the predefined rules and variables, we can omit them for a shorter mkfile:

```
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o:    prog.h
c.o:    prog.h
```

Rules with no prerequisites

Rules need not actually build their targets. Some rules are simply shell scripts embedded in the mkfile for convenience. For example, most mkfiles have the target *clean :*

```
clean:
        rm -f *.o prog core
```

Note that *clean* is intended as a label, not a file. Unfortunately, if a file named *clean* exists, the recipe will not be executed, since *clean* is up to date (because no prerequisite has caused it to be out of date). We want to avoid any such inadvertent interactions with the file system. **mk** allows a label to have an attribute of *virtual* , which means that it is distinct from a file of the same name. Targets can be marked as virtual by appending a *V:* to the colon separator between targets and prerequisites:

```
clean:V:
        rm -f *.o prog core
```

Attributes (like *V*) that apply to labels can be given by more than one rule. Where feasible, it is good style to use an attribute on all rules applying to a particular label, not just one. Other attributes are described below.

Rules with multiple targets

The rules relating *b.o* and *c.o* to *prog.h* can be combined into one rule with two targets.

```
CFLAGS=-g -p
prog:  a.o b.o c.o
       $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
clean:V:
       rm -f *.o prog core
```

If a rule with multiple targets has no recipe, it is simply a shorthand notation for all the simple rules with one target. A rule with multiple targets and a recipe has subtle implications described below. To motivate the subtleties, we digress to describe the yacc parser generator.

yacc takes a file describing a grammar and produces the source for a C routine that will parse input according to the given grammar. The source is put in the file *y.tab.c* . **yacc** also produces a header file called *y.tab.h* that links the parser to a lexical analyzer. The grammar file also contains semantic action code. Typically, changes to the grammar file do not change the header *y.tab.h* , but only the semantic routines.

Let us add a grammar and a lexical analyzer to *prog*:

```
prog:  a.o b.o c.o y.tab.o lex.o
       $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
lex.o: y.tab.h
y.tab.c y.tab.h:    gram.y
       yacc -d gram.y
```

The grammar is kept in *gram.y* (the conventional suffix for **yacc** input is *.y*). The *-d* option to **yacc** produces *y.tab.h* . Unfortunately, this mkfile does too much work in the normal case. Every time the grammar file is changed, a new *y.tab.h* is made and thus *lex.o* will always be out of date even though the contents of *y.tab.h* may not have been changed. The best solution maintains another header file (say *x.tab.h*) that only changes when necessary, that is, when the contents of *y.tab.h* actually change. The new mkfile is

```
prog:  a.o b.o c.o y.tab.o lex.o
   $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
lex.o: x.tab.h
x.tab.h:      y.tab.h
   cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
y.tab.c y.tab.h:    gram.y
   yacc -d gram.y
```

The recipe for *x.tab.h* is a conditional shell construct; if the command *"cmp -s x.tab.h y.tab.h"* returns with an error (the files are different), then execute the command *"cp y.tab.h x.tab.h"* to copy *y.tab.h* onto *x.tab.h* . In the case where *y.tab.h* doesn't change, the action is straightforward:

```
$ mk -e
y.tab.c(541051073) < gram.y(541051092)
y.tab.h(541051072) < gram.y(541051092)
yacc -d gram.y
y.tab.o(541051082) < y.tab.c(541051100)
cc  -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051099)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp not done
prog(541051087) < y.tab.o(541051109)
cc  -o prog a.o b.o c.o y.tab.o lex.o
$
```

If we now change the grammar so that the header file does change:

```
$ mk -e
y.tab.c(541051100) < gram.y(541051148)
y.tab.h(541051099) < gram.y(541051148)
yacc -d gram.y
y.tab.o(541051109) < y.tab.c(541051155)
cc  -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051154)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp done; x.tab.h updated
lex.o(541042267) < x.tab.h(541051165)
cc  -c lex.c
prog(541051114) < y.tab.o(541051163)
prog(541051114) < lex.o(541051169)
cc  -o prog a.o b.o c.o y.tab.o lex.o
$
```

The subtleties are twofold. The first is that the time stamps for files are only examined when the file is initially referenced or when it is the target of a rule. If *y.tab.h* had not been a target for the **yacc** rule, then **mk** would assume that *y.tab.h* had not been updated. The second subtlety is that the rule for *x.tab.h* need not change *x.tab.h* . If it does not, then *lex.o* need not be recompiled.

Aggregates

Some of the things we would like to maintain with **mk** are actually collections or *aggregates* of entities, such as UNIX System object libraries (archives maintained by **ar**). Other (unsupported as yet) examples are **cpio** and SCCS files. The type of aggregate is determined by the file's "magic number." Each type has support code within **mk** to get the time stamp of a member and to "touch" (see below) a member. The notation *a(m)* refers to member *m* of aggregate *a* . For example, consider an archive *lib.a* made up of *a.o* , *b.o* , and *c.o* . The mkfile looks like

```
lib.a:N:      lib.a(a.o) lib.a(b.o) lib.a(c.o)
lib.a(%.o):   %.o
        ar r lib.a $stem.o
```

(The *N* attribute is described below.) As each new *.o* file is generated, it is put into *lib.a*. This is straightforward and correct but inefficient: an **ar** command is executed for every out of date object file. A better way is to generate all the *.o* files and then do the *ar* . The new mkfile relies on a shell script called *membername :*

```
lib.a: lib.a(a.o) lib.a(b.o) lib.a(c.o)
        ar r lib.a `membername $newprereq`
lib.a(%.o):N: %.o
```

Membername takes aggregate notation and extracts the member names. For example,

```
$ membername 'lib.a(a.o)' 'lib.a(b.o)'\
a.o b.o c.o
$
```

The quotes are to stop the shell from interpreting the *()* . We use the variable *newprereq* (supplied by **mk**) because we only need to replace the object files that have changed.

Targets without recipes

Sometimes a target may be out of date with respect to some prerequisites and yet there is no recipe that **mk** can use to remake the target. For example,

```
all:V: prog1 prog2 prog3
```

When we say *"mk all"* , we simply want to make *prog1* , *prog2* , and *prog3* . Because *all* is virtual, it doesn't matter that it isn't actually remade; all **mk** does with virtual targets is propagate the date stamp. A more serious example is

```
prog:  a.o b.o c.o
       $CC $CFLAGS -o $target $prereq
%.o:   hdr.h
```

If there is no *c.c* , then **mk** does not know how to make *c.o* and yet needs it in order to make *prog* . The only rule for *c.o* is the dependency on *hdr.h* which has no recipe. Therefore, in these situations, where a nonvirtual target is out of date but none of the rules applying to that target has a recipe, **mk** will stop and complain that it doesn't know how to make that target.

There is one type of rule where this is not the right thing to do. Take for example the above archive rule:

```
lib.a: lib.a(a.o) lib.a(b.o) lib.a(c.o)
       ar r lib.a 'membername $newprereq'
lib.a(%.o):N: %.o
```

There is no recipe to build (say) *lib.a(a.o)* but this is okay as it will be built later by *ar* . The *N* attribute stops **mk** from complaining that there is no recipe.

Parallel processing

mk executes recipes by continually traversing the dependency graph looking for targets that can be made. For example, in our mkfile:

```
prog:  a.o b.o c.o y.tab.o lex.o
   $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
lex.o: x.tab.h
x.tab.h:      y.tab.h
   cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
y.tab.c y.tab.h:    gram.y
   yacc -d gram.y
```

the target *a.o* can be made immediately, while the target *y.tab.o* has to wait for *y.tab.c* to be made. When **mk** finds a recipe it can execute, it puts the recipe on a queue. When the recipe terminates, **mk** updates the dependency graph. The number of recipes executing simultaneously is bounded by the value of the variable *NPROC* , which is

initially one. On multi-processor machines, **mk** goes faster with higher values; most mkfiles on our 12 processor machine have *NPROC* between 6 and 10. In most situations, increasing *NPROC* beyond a certain limit gains almost nothing. The other way to speed up parallel builds is to ensure that as many recipes as possible are executing; that is, order the sub-targets such that the slowest are done first. While **mk** gives no guarantees about the order of builds, generally prerequisites are built in left-to-right order as in the mkfile.

The *-u* (utilization) option measures how many seconds (real time) are spent with so many recipes executing. For example, building *prog* with three simultaneous recipes yields

```
0: 1
1: 4
2: 7
3: 10
```

This means that the entire run took 22 seconds real time; 10 seconds with three recipes running, 7 with two and 4 with one. The time with zero recipes executing corresponds to **mk** reading the mkfile and building the dependency graph.

Parallel execution implies that recipes should not interact unnecessarily. For example, the first version of the library mkfile should not be run in parallel as simultaneous *ar 's* on the same archive interfere. The second version can be run in parallel because only one **ar** is done, after all the object files are made.

Missing intermediates

In all the examples we have seen so far, **mk** has made all the targets "between" the file that changed and the main target. This is not always done. Any non-existent intermediate target (a target other than the root target with prerequisites) is treated specially. If pretending it existed with the time stamp of its most recent prerequisite would make all targets that depended on it be up to date, then it is not made. For example, in our mkfile:

```
$ mk -e
mk: 'prog' is up to date
remove a.o
$ mk -e
pretending a.o has time 540869454
mk: 'prog' is up to date
```

The intuition is that if we use the mkfile to build the targets, then removing the intermediates causes no harm. Of course, if we actually need the missing intermediates, **mk** builds them.

```
change b.c
$ mk -e
pretending a.o has time 540869454
b.o(540869546) < b.c(541350226)
cc  -c b.c
unpretending a.o because of prog because of b.o
a.o(0) < a.c(540869454)
cc  -c a.c
prog(541104056) < a.o(541350255)
prog(541104056) < b.o(541350244)
cc  -o prog a.o b.o c.o y.tab.o lex.o
$
```

The action is not too hard to follow: first **mk** sees that *a.o* is missing and pretends it is there. Then **mk** notices *b.o* is out of date and needs to be rebuilt. When *b.o* is finally built, it causes *prog* to become out of date and therefore **mk** no longer can pretend that *a.o* is up to date. It then builds *a.o* and then *prog* .

The major advantage of missing intermediates is avoiding multiple copies of files. For example, in our mkfile to maintain a library, we keep two copies of every object file. By using the notion of missing intermediates, we can keep one copy — the copy we need in the archive. To do so, simply remove the object files after they have been archived:

```
lib.a: lib.a(a.o) lib.a(b.o) lib.a(c.o)
       names='membername $newprereq'
       ar r lib.a $names && rm $names
lib.a(%.o):N: %.o
```

We store the object files' names in the variable *names* to avoid executing *membername* twice. The *&&* is another conditional shell construct; we remove the files only if the archive command succeeds.

The special treatment of missing intermediates is suppressed by the *-i* option of **mk**.

Administrative

mk provides an easy way to bring a target up to date without actually doing any work. For example, if we change *prog.h* in such a way that *b.o* or *c.o* won't change (such as adding a comment), we don't want to recompile the files. Instead, we can ask **mk** to modify the files' time stamps.

```
add something to prog.h
$ mk -t
touch(b.o)
touch(c.o)
touch(prog)
$
```

mk lists the files it modified. This is a dangerous feature; use it carefully and sparingly. Virtual targets are not affected because *touch*ing only changes files.

mk can also tell us what it would do without actually doing it. The option *-n* causes recipes to be printed rather than executed. There are two main problems. **mk** assumes that every recipe will update all its targets. Normally this is true, but for our mkfile, *"mk -n"* would erroneously indicate that *lex.o* will always be remade. Thus, unnecessary work may be indicated. The second problem is that when **mk** prints the recipe it is about to execute (or would execute under *-n*), it expands recognizable references to shell variables. It does this without parsing the shell script and thus can make mistakes by ignoring variable assignments within a recipe and with constructs like *for* loops. For example, with the mkfile (the *Q* attribute suppresses the normal recipe echo)

```
i=a b c
all:Q:
        for i in x y z
        do
                echo $i
        done
```

the difference between **mk** and *"mk -n"* is:

```
$ mk -n
for i in x y z
do
        echo a b c
done
$ mk
x
y
z
$
```

This latter problem applies to the normal recipe echo as well.

Sometimes we would like to know what **mk** would do if some files were changed. The *-wfiles,...* option supports this "what if" query by setting the time stamps internally for the named files to the current time. With our mkfile for *prog* , we can ask what would happen if we changed *prog.h* :

```
$ mk -n -wprog.h
cc  -c b.c
cc  -c c.c
cc  -o prog a.o b.o c.o y.tab.o lex.o
$
```

The advantage of *-w* is that neither the files nor their time stamps are changed. Of course, *-w* can be used without using *-n* . For example, to force **mk** to remake *b.o* we can say

```
$ mk -wb.c b.o
cc  -c b.c
$
```

Quoting

The quoting rules for assignment lines and rule header lines are intended to be the same as for **sh (1)** (the Bourne shell). As these rules are not described clearly in **sh (1)**, we describe **mk** 's quoting rules below. The term *"quoting a character"* means making that character stand for itself, rather than any special, or meta, meaning. For example, *$a* stands for the value of the variable *a* , whereas *\$a* (the *$* is now quoted) stands for

the two characters *$* and *a* .

Input is collected until a newline without a preceding \ is seen. During this collection, \ quotes every character except newline (which is deleted) and text between unquoted single quotes is quoted. Inside single quotes, every character is quoted. The resulting text is rescanned for unquoted backquotes. When an unquoted backquote ‘ is seen, input is collected until the next unquoted backquote. The collected input is given as standard input to the shell and the standard output replaces the collected input and the two backquotes as quoted text. The resulting text is rescanned (again!) for double quotes and variable expansions. Text between double quotes is quoted after variable expansion is done and \ only quotes the characters *#'$* .

The text is then broken into parts separated by unquoted white space and any part containing *[*?* as unquoted characters is then expanded as filenames as per **sh (1)**.

More on metarules

Certain metarules can lead to infinite dependency graphs. For example, the metarule

```
%:      %.z
        unpack $stem.z
```

gives this dependency graph

```
x  --->  x.z  --->  x.z.z ---> x.z.z.z ...
```

The problem arises any time a metarule has a prerequisite that can be a target of the same rule. **mk** handles this problem by restricting the number of times a metarule is used in generating prerequisites to the value of the variable *NREP* . This value is normally one; if set to 3, the dependency graph for *x* in our example is

```
x  --->  x.z  --->  x.z.z  --->  x.z.z.z
```

Thus, setting *NREP* to greater than one is necessary if we have files that have been packed repeatedly.

Getting Fancy

This section is a collection of problems **mk** users (and **make** users in general) run into and how I have seen **mk** used to deal with those problems. In general, other **make** programs have special features to deal with these problems; **mk** solves them, sometimes clumsily, by its general mechanisms.

One of the first problems you run into when the mkfiles get big is having several versions of file names. For example:

```
SRC=main.c lex.l gram.y subr.c
OBJ=main.o lex.o gram.o subr.o
pr:V:  $SRC
       pr $prereq | lp
prog:  $OBJ
       $CC $CFLAGS -o $target $prereq
```

We can derive the *OBJ* list from the *SRC* list by:

```
SRC=main.c lex.l gram.y subr.c
OBJ=`echo "$SRC" | sed 's/\../.o/g'`
```

Often, an object file generated from a C source depends on compiler flags embedded in the makefile. The "correct" thing to do would be to make the object file depend on the makefile but this would imply remaking all the objects whenever you changed the makefile even though most of the changes would not be concerned with the compiler flags. Some **make**s, for example **nmake,** store the flags used to make the object in a database and thus can tell when it changes. The **mk** solution is to follow the paradigm of files depending on files; put the C compiler flags in a file!

```
$ cat CFLAGS
-g -DSYSV
$ cat mkfile
CFLAGS=`cat CFLAGS`
%.o:   CFLAGS
```

Everything falls out; objects get remade just when they need to. This trick can often be used when files depend on parts of the mkfile. Other examples are adding a new object file to a program or deleting an element of an archive library. Both of these cases can be handled the same way; **make** them depend on a file which contains the element names.

Sometimes a recipe fails *after* modifying the target. Often in this case, the fix doesn't cause the target to become out of date. (This can happen when **mk** is interrupted; a target is not complete but it is up to date!) For these cases, use the *D* attribute; **mk** will remove the target on errors. This is normally only necessary for recipes of the form

```
.... > $target
```

mk supports a general mechanism for determining if a file is out of date. The *:Pcmd:* attribute means that to find out if *a* is out of date with respect to *b* , **mk** runs the command

```
cmd 'a' 'b'
```

and uses the exit status as the out of date indicator. For example, the yacc rule can be rewritten as

```
x.tab.h:Pcmp -s:    y.tab.h
        cp $prereq $target
```

The meaning here is that *x.tab.h* is out of date only if it is different from *y.tab.h* .

Often mkfiles do not include all the header file dependencies. How do you safely remake things after changing the definition of a variable *var* ?

```
$ mk -w"`grep -l var *.[cyl]`"
```

The double quotes keep the file names together as one argument.

mk does not support the conditional variable reference in **sh (1).** You can easily fake it with backquotes:

```
GOAL=`echo "${GOAL:-all}"`
```

This sets *$GOAL* to the value in the environment and if that is null, set it to *all* .

Differences between **make** and **mk**

The qualitative differences between **mk** and **make** can be summarized as (pertinent differences to the older **make s** are noted in parentheses):

- **make** builds targets when it needs them, allowing systematic use of side effects. **mk** constructs the entire dependency graph before building any target.
- **make** supports suffix rules and % metarules **mk** supports % and regular expression metarules. (Older **make s** only have suffix rules.)

- **mk** performs transitive closure on metarules, **make** does not.
- **make** supports cyclic dependencies, **mk** does not.
- **make**'s recipes are collections of one-line shell commands, executed a line at a time. Variable values are passed by editing the recipe text before passing it through to the shell. **mk** 's are simply shell scripts executed as one unit. Variable values are passed through environment variables.
- **make** supports parallel execution of single-line recipes when building the prerequisites for specified targets. **mk** supports parallel execution of all recipes. (Older **make** s did not support any parallel execution of recipes.)
- **make** uses special targets (beginning with a .) to indicate special processing. **mk** uses attributes indicated by qualifiers after the : separator in a rule definition.
- **mk** allows the standard output of a recipe to be read as an additional mkfile while **mk** is running. This allows a mkfile to configure itself at run time.
- **mk** supports *virtual* targets which exist only within an execution of **mk** and are independent of the underlying file system.
- **mk** supports a general mechanism for deciding whether a file is out of date as well as the normal method of comparing file modification times.

In most situations, mkfiles and makefiles (the input for **make**) will have only minor syntactic differences. In practice, mkfiles are often bigger because of embedded shell scripts or to make the most of underlying parallel hardware. (Parallelism works best when it has a lot to do; one way to do this is to merge mkfiles together and have one mkfile to control multiple subdirectories.)

The most striking difference between **mk** and **make** is in speed of execution. There are three main factors involved. **make** uses a linear list to access variables and rules; **mk** uses a hash table. **mk** and **make** use time stamps in slightly different ways; **make** often has to measure a file's time stamp unnecessarily. If there are metarules, **mk** will typically create a much larger dependency graph than **make** . The graph gets pruned but at the cost of testing (for existence) a large number of files. In the examples given below, execution times are given (in seconds) as a sum of user time (a measure of how efficiently the dependency graph is built and executed) and system time (a measure of how many time stamps are measured). The times do not include times for recipe executions.

For mkfiles with no metarules, **mk** is always faster than **make** because of better accessing algorithms. For example, the mkfile to compile the operating system describes 83 object files. **make** takes 19.8u+3.6s (19.8 seconds of user time, 3.6 seconds of system time), **mk** takes 6.6u+3.6s. **mk** is faster by a factor of 3 (user time) and 2.3 (user+sys).

For more normal mkfiles (that use the builtin metarules), **make** is somewhat faster than **mk** until about a dozen prerequisites are involved. **mk** is much better for larger mkfiles. In most cases, **mk** 's performance can be improved by only using necessary metarules. For example, for a program made from 61 object files all compiled from *.c* files, we give the times for a normal mkfile and a mkfile that has only one metarule (generating *%.o* from *%.c*).

Command	Run Time	Relative Speed (user)	Relative Speed (user+sys)
make	12.0u+9.7s	1	1
mk (all metarules)	5.1u+4.0s	2.3	2.4
mk (one metarule)	3.9u+2.9s	3	3.2

mk handles aggregates efficiently. The main C library has 242 members. **make** takes 47.7u+10.9s, **mk** takes 6.3u+12.5s. **mk** is faster by a factor of 7.6 (user time) and 3.1 (user+sys).

The final example comes from Ted Kowalski at AT&T Bell Laboratories. The mkfile is about 20,000 characters and describes an experimental workstation environment built from 238 .c files, 59 .h files, 7 .y files and 7 .l files. The mkfile makes heavy use of variables. **make** takes 278.8u+16.2s, **mk** takes 8.4u+10.5s. **mk** is faster by a factor of 33 (user time) and 15.6 (user+sys).

Despite the marked speed advantage of **mk** over **make** , the main reason users in our computing community use **mk** is its functionality, in particular, transitive closure on metarules, parallel execution of recipes, and the regular expression metarules.

Conversion from make to mk

Conversion of makefiles into mkfiles comes in two parts. The first is a mechanical process of syntax conversion (such as changing variable references) handled by the **sed** (1) script **mk**conv . It produces a mkfile on its standard output and warns about known nasties such as recursive calls to **make** . For routine makefiles, this is all that needs to be done.

If needed, the other changes have to be done by hand. They involve the use of side-effects by **make** , such as the normal way **yacc** grammars are handled. The proper way to handle these grammars is described above; in other cases, the general rule is to tell the truth about dependencies and let the dynamic time measuring prevent unnecessary work. **mk** has much support for the debugging for these cases, particularly where the makefile is complex or subtle. The most useful options are *-dg* (to find out the exact dependency graph), *-e* (to explain why **mk** thinks something is out of date), and *-n* and *-w* (to conduct what if? experiments).

Availability of mk

mk can be obtained in unsupported source form from the AT&T UNIX System Toolchest (1-201-522-6698 to talk to a person; 1-201-522-6900, login *guest* , to browse and talk to a computer). **mk** runs on a variety of hardware: VAX, 3B, Sun (68000), Sequent (32032) and Cray X-MP/24. It is not tuned to any particular variant of the UNIX operating system; it runs on Research V9, BSD4.[23], Dynix, Cray and various forms of System V. The distribution includes the **mk** source, library support for systems without the V9 libraries and documentation including a manual entry and this tutorial.

Bugs, questions or other feedback should be directed to the author via the UNIX System Toolchest Administrator.

The Principles

mk 's semantics and syntax were designed according to a few general principles or guidelines. The syntax of mkfiles is almost exactly the same as a makefile (used by **make**). (The only syntactic change for rules is the attribute marking.) **mk** 's variables are exactly the same as shell variables. Recipes are written in **sh (1),** not a special purpose language. The regular expression syntax and semantics were adopted from existing tools (such as **egrep** and **ed**), trading some awkwardness for familiarity. **make** 's metarules (already a generalization of the early **make** suffix rules) were extended to full regular expressions. **mk** performs the transitive closure on the target-prerequisite relations defined by all rules, including metarules. The primitive form of parallel processing supported by **make** has been generalized to allow parallel execution of any recipe. By constructing the entire dependency graph before executing any recipes, **mk** maximizes the benefits from parallel processing. **make** 's variables and recipes were so close to being shell variables and scripts that the differences were removed in **mk** . Making recipes shell scripts had the further advantage that **mk** does not have to parse or process the recipes. The use of special target and prerequisite names (beginning with a dot) to indicate special actions has been dropped in favor of a more explicit notion of target *attributes* . Recent versions of **make** (such as **nmake**) focus on the issues connected with building software and generally contain much builtin knowledge about C programming. **mk** , on the other hand, is a tool for maintaining file dependencies, whether they be programs or circuit board descriptions. It offers general purpose and powerful mechanism for all users, not just help for programmers. And it does so by having a simple, comprehensible model of behavior that a user can predict without having to be an expert.

Appendix 1

The default variable definitions are:

```
AS=as
CC=cc
CFLAGS=
FC=f77
FFLAGS=
LDFLAGS=
LEX=lex
LFLAGS=
NPROC=1
NREP=1
YACC=yacc
YFLAGS=
```

The builtin rules are

```
%.o:    %.c
        $CC $CFLAGS -c $stem.c
%.o:    %.s
        $AS -o $stem.o $stem.s
%.o:    %.f
        $FC $FFLAGS -c $stem.c
%.o:    %.y
        $YACC $YFLAGS $stem.y &&
        $CC $CFLAGS -c y.tab.c && mv y.tab.o\
         $stem.o; rm y.tab.c
%.o:    %.l
        $LEX $LFLAGS -t $stem.l > $stem.c &&
        $CC $CFLAGS -c $stem.c && rm $stem.c
```

The environment for the recipe's shell is augmented by these variables:

alltarget	all the targets for this rule.
newprereq	the prerequisites that are more recent than the target.
nproc	this is the process slot for this recipe. It is a number between zero and *$NPROC-1* inclusive. It is useful for parallel execution on a single CPU machine on a network.

pid	the process id for the **mk** invoking this script. This is useful for communicating with other rules.
prereq	all the prerequisites for this target. This may include prerequisites from several rules.
stem,...	the value of % in a metarule. It is null for a non-metarule. The value of the *n th* subexpression in a regular expression metarule is put in the variable *stemn* , for *n* <*10*. It is null otherwise.
target	the targets being built for this rule.

Appendix 2

The changes (in no particular order) are:

1. The order of processing the environment variables has changed. The original hope was that if the environment overrode the values inside the mkfile, there would be no need for command line variable assignments. This didn't work out. In fact, the chief offender was **mk** calling itself recursively (all the variables were now in the environment!). The processing order is now the same as **make** .

2. The builtin rule for **yacc** grammars was tweaked slightly so as to not rely on the V9 **yacc** .

3. The filenames for the *-w* option can be separated by blanks and newlines as well as commas. This allows the names to be generated conveniently by a backquote expression.

4. The *D* attribute was introduced for the real problem of recipes which update their targets even though they fail. I resisted this as long as I could; **mk** is supposed to create files, not remove them.

5. The *N* attribute is a sign of **mk** 's increasing dogma. The problem is rules which have out of date prerequisites but no recipe to update the target. If the target is virtual, this is benign and **mk** does not complain. Otherwise, the target is being made as a side-effect or is not being made. Sometimes, as in the library rules, the former is true. Other times, the latter is true. For example,

```
prog:  a.o b.o c.o
       $CC $CFLAGS -o $target $prereq
%.o:   hdr.h
```

   If we add a new object to the list (say *d.o*) without creating a corresponding *d.c* , then **mk** used to happily say everything was fine and the loader would complain about a missing *d.o* ! I regard allowing a recipe to run without all of its prerequisites made as a serious error so **mk** doesn't any more. (The reason **mk** didn't complain about not knowing how to make *d.o* was because of the *"%.o:hdr.h"* rule.)

6. The *P* attribute was supposed to be in the first release but I couldn't figure out how to describe the semantics. The key is that out-of-date-ness and date stamp propagation are separate issues. So that even if files are related by a general nontemporal relation, the date stamps can be propagated in the normal way and allow proper interaction with targets made with the normal mechanism.

The resulting rule for updating *x.tab.h* in the **yacc** example is unexpectedly nice.

7. A few small bugs were fixed, the main one being that backquote expressions were evaluated before rather than after processing single quotes.

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *BSD UNIX Programmer's Manual*
Order No.017272-A00

Your Name _____ Date _____

Organization _____

Street Address _____

City _____ State _____ Zip _____

Telephone number (____) _____

When you use the Apollo system, what **job(s)** do you perform?

- ☐ Programming
- ☐ Hardware Engineering
- ☐ Other (describe)_____
- ☐ Application End User
- ☐ System Administration

How many years of **experience** do you have in using the Apollo system:

_____

What programming **languages** do you use with the Apollo system?

_____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: _____

_____

_____

No postage necessary if mailed in the U.S.

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *BSD UNIX Programmer's Manual*
Order No.017272-A00

Your Name _____ Date _____

Organization _____

Street Address _____

City _____ State _____ Zip _____

Telephone number (____) _____

When you use the Apollo system, what **job(s)** do you perform?

- ☐ Programming
- ☐ Hardware Engineering
- ☐ Other (describe)_____
- ☐ Application End User
- ☐ System Administration

How many years of **experience** do you have in using the Apollo system:

_____

What programming **languages** do you use with the Apollo system?

_____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: _____

_____

_____

No postage necessary if mailed in the U.S.

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold