ROSCOE USER GUIDE

Version 1.0

by

Ron Tischler
Marvin Solomon
Raphael Finkel

Computer Sciences Technical Report #336

September 1978

ROSCOE USER GUIDE

Version 1.0
September 1978

Ron Tischler
Marvin Solomon
Raphael Finkel

Technical Report 336

Abstract

Roscoe is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. This document describes Roscoe from the viewpoint of a user or a writer of user-level programs. All system service calls and library routines are described in detail. In addition, the command-line interpreter and terminal input conventions are discussed. Companion reports describe the purposes and concepts underlying the Roscoe project and give detailed accounts of the Roscoe utility kernel and utility processes.

TABLE OF CONTENTS

ROSCOE USER GUIDE

1. INTRODUCTION

Roscoe is an experimental operating system for controlling a network of microcomputers. It is currently implemented on a network of five Digital Equipment Corporation LSI-11 computers connected by medium-speed lines.* The essential features of Roscoe are:

1. All processors are identical. However, they may differ in the peripheral units connected to them. Similarly, all processors run the same operating system kernel.

2. No memory is shared between processors. All communication is done by explicit passing of messages between physically connected processors.

3. No assumptions are made about the topology of interconnection except that the network is connected (that is, there is a path between each pair of processors). The lines are assumed to be sufficiently fast that fairly tight interaction is possible between processes on different machines.

4. The network should appear to the user to be a single machine. A process runs on one machine, but communicating processes have no need to know and no way of finding out if they are on the same processor.

----

*This equipment was purchased with funds from National Science Foundation Research Grant #MCS77-08968.

1.1 Purpose of this Document

This document describes Roscoe from the point of view of a user or user-programmer. It is both a tutorial and a reference guide to the facilities provided to the user. All information necessary to the programmer of applications programs should be found here.

Further discussion of the concepts and goals of Roscoe are discussed in [Solomon and Finkel 78]. That document also lists some research problems that the Roscoe project intends to investigate. The operating system kernel that provides the facilities listed below is described in considerable detail in [Finkel and Solomon 78]. Similar detailed documentation about utility processes (such as the File System Process, the Teletype Driver, the Command Interpreter, and the Resource Manager) is contained in [Tischler, Finkel, and Solomon 78].

Roscoe has been developed with extensive use of the UNIX operating system [Ritchie and Thompson 74]. All code (with the exception of a small amount of assembly language) is written in the C programming language [Ritchie 73]. The reader of this document is assumed to be familiar with both UNIX and C.

A new programming language is being designed for applications programs under Roscoe; it will be described in a future report. However, this version of the Roscoe User Guide assumes that all Roscoe software is written in C.

1.2 <u>Caveat</u>

Roscoe is in a state of rapid flux. Therefore, many of the details described in this Guide are likely to change. The reader who intends to write Roscoe programs should check with one of the authors of this report for updates.

1.3 <u>Format of this Guide</u>

Section 2 provides an overview of the concepts and facilities of Roscoe. It is organized according to general subject areas. Specific functions are mentioned but not described in full detail. Section 3 is a programmer's reference manual. Each function is listed alphabetically, its syntax and purpose are described, and it is classified as a Service Call (an invocation of an operating system kernel routine) or a Library Routine (a procedure linked into the user program). Section 4 describes the command line interpreter and lists the commands that may be entered from the terminal. Section 5 describes the conventions governing terminal input/output. Section 6 presents protocols for communicating with the various utility processes.

2. ROSCOE CONCEPTS AND FACILITIES

The fundamental entities in Roscoe are: <u>files</u>, <u>programs</u>, <u>core images</u>, <u>processes</u>, <u>links</u>, and <u>messages</u>. The first four of these are roughly equivalent to similar concepts in other operating systems; the concepts of links and messages are idiomatic to Roscoe. A file is a sequence of characters on disk. Each file has directory information giving the time of last modification and restrictions on reading, writing, and execution. The contents of a file may contain header information that further identifies it as an executable program. Version 1 of Roscoe uses the UNIX file system; therefore, the reader familiar with UNIX should have no problem understanding Roscoe files.

Program files contain *text* (machine instructions), *initialized data*, and a specification of the size of the uninitialized global data space (*bss*) required by the program. Program files also contain relocation information and an optional symbol table.

A *process* is a locus of activity executing a program. Each process is associated with a local data area called its *stack*. A program that never modifies its global initialized or bss data but only its local (stack) data is *re-entrant*, and may be shared by several processes without conflict. A main-storage area containing the text of a program, its initialized data, and a bss data area, but not including a stack, is called a *core image*. The initiation of a process entails locating or creating (by *loading*) a core image, allocating a stack, and initializing the necessary tables to record its state of execution. Similarly, when a process *dies*, its tables are finalized and its stack space is reclaimed. If no other processes are executing in its core image, then the space occupied by the core image is available for re-use.

Some processes, called *utility processes*, provide facilities to other processes, such as device or file management. Utility processes may invoke service calls not intended to be used by the casual user, but otherwise they behave exactly like user processes.

A _link_ combines the concepts of a communications path and a "capability." A link represents a logical one-way connection between two processes, and should not be confused with a _line_, which is a physical connection between two processors. The link concept is central to Roscoe. It is inspired and heavily influenced by the concept of the same name in the Demos operating system for the Cray-1 computer [Baskett 77]. Each link connects two processes: the _holder_, which may send messages over the link, and the _owner_, which receives them. The holder may duplicate the link or give it to another process, subject to restrictions associated with the link itself. The _owner_ of a link, on the other hand, never changes.

Links are created by their owners. When a link is created, the creator specifies a _code_ and a _channel_. The kernel automatically tags each incoming message with the code and channel of the link over which it was sent. Channels are used by a process to partition the links it owns into subsets: When a process wants to receive a message, it specifies a set of channels. Only a message coming over a link corresponding to one of the specified channels is eligible for reception. A link is named by its holder by a small positive integer called a _link number_, which is an index into a table of currently-held links maintained by the kernel for the holder. All information about a link is stored in this table. (No information about a link is stored in the tables of the owner.)

A _message_ may be sent by the holder to the owner of a link. In addition, certain messages are manufactured by the kernel to inform the owner of a link of changes in its status. For example, the creator of a link may specify that when the link is destroyed, a DESTROYED notification be sent along it. Such messages are identified to the recipient by an unforgeable field.

A message may contain, in addition to MSLEN (currently 40) characters of text, an _enclosed link_. The sender of the message specifies the link number of a link it currently holds. The kernel adds an entry to the link table of the destination process and gives its link number to the recipient of the message. In this way, the recipient becomes the holder of the enclosed link. If the original link is not destroyed, the sender and the recipient hold identical copies of the link.

There are two kinds of links: _request_ and _reply_. A reply link is distinguished by the fact that it can only be used once; it is destroyed when a message is sent over it. A reply link may not be the enclosed link in a message sent over another reply link. Similarly, a request link cannot be sent over a request link. These restrictions enforce a communication protocol in which one process does most of the talking, over a REQUEST link, and can be answered once for each enclosed REPLY link.

The remainder of this section lists service calls and library routines by subject area.

2.1 Links and Messages

A new link is created by a process through the "link" service call. Initially, the creator is both holder and owner of the link. Messages are sent with the "send" service call, which specifies a link over which the message is to be sent, the message text and an optional enclosed link. Messages are accepted by "receive," which specifies a set of channels, a place to put the message, and a maximum time the recipient is willing to wait. "Receive" can also be used to sleep a specified period of time by waiting for a message that will never arrive. A simple send-receive protocol is embodied in the library functions "call" and "recall," which are simpler to use than send and receive, and should be adequate for most routine communication.

2.2 Processes

A process may spawn others by communicating with the Resource Manager; typical cases are handled by "fork". When calling "fork", the parent may indicate a link that it wishes to give to the child; the child obtains this link with "parline". In certain cases the parent can kill the child with "killoff"; in other cases a control-C entered at the terminal can have this effect.

Every user process is born holding link number 0, whose destination is the Resource Manager on that process's machine. A process can terminate itself by calling "die"; and can sleep by using either "nice" or "receive".

The service calls "load", "startup", "kill", and "remove" control core images and processes. They are used by the Resource Manager and are not intended for the typical user.

2.3 Timing

Roscoe has two notions of time. One is the wall clock, which keeps track of seconds in real time. Messages sent between Resource Managers are routinely used to keep the various machines synchronized. There is also an interval timer, which may be used to monitor elapsed time in increments of ten-thousandths of seconds. No process may change the interval timer.

The wall clock is referenced, changed, enciphered, and deciphered by "date", "setdate", "datetol", and "ltodate", respectively. The interval timer is referenced by "time".

2.4 Interrupt Level Programming

User programs may handle their own interrrupts. A process may establish an interrupt-level routine with the "handler" call. The interrupt-level routine should, of course, be thoroughly debugged and fast. Interrupt-level routines may notify the process that established them by calling "awaken"; the process to be notified uses "receive" to obtain this notification.

Only the Teletype Driver uses this feature.

2.5 Input/Output

To use files, a process first obtains a link to the File System Process by calling "fsline". This link is used in subsequent "create", "open", "stat", "alias", and "unlink" calls, which behave much like the UNIX calls with similar names. "Open" and "create" calls return links to be used for performing "stat", "close", and all input/output operations on the open file.

To use the terminal, a process obtains input and output links by calling "inline" and "outline", respectively. An input link can be used to discover or change terminal modes (only the Command Interpreter uses this feature) and to perform terminal input. An output link can be used for terminal output. These links may also be "closed"; they are closed automatically when a process dies. The Teletype Driver allows at most one input link to be open at a time.

Reading is performed by the routines "read" and "readline". Writing is performed by "write" and, if formatting is desired, by "print". The service call "printf" is identical to "print" except that it does direct terminal output; it is a debugging tool not intended for the typical user.

Reads and writes are not more efficient with buffers of size 512, because Roscoe splits up I/O into packets of size MSLEN bytes anyway.

2.6 Miscellaneous Routines

The following routines from the C library also exist in the Roscoe library: atoi, long arithmetic routines, reset, setexit, strcopy, streq, strge, strgt, strle, strlen, strlt, strne, and substr.

An additional routine supplied by Roscoe is "copy".

2.7 Preparing User Programs

User programs for Roscoe are written in the C programming language. They are compiled under UNIX on the PDP-11/40 in the directory "/usr/network/roscoe/user" and should include the files "user.h" and "util.h". Source programs should have filenames ending with ".u". To prepare a file named "foo.u", execute "makeuser foo", which creates an executable file for Roscoe named "foo".

3. ROSCOE PROGRAMMER'S MANUAL

The following is an alphabetized list of all the Roscoe service calls and library routines.

3.1 Awaken (Service Call)

awaken()

Only an interrupt-level routine may use this call. It sends a message to the process that performed the corresponding "handler" call along the channel specified by that "handler" call.

Returned values: Success returns a value of 0. -2 is returned if the message cannot be sent because no buffers are available; an "awaken" may succeed later.

3.2 Call (Library Routine)

int call(ulink,outmess,inmess) char *outmess,*inmess;

This routine sends a message to another process and receives a reply. The link over which the message is sent is "ulink", which should be a REQUEST link. The argument "outmess" points to the message body to be sent, of size MSLEN. Similarly, "inmess" points to where the reply body, of size MSLEN, will be put. If "inmess" is 0, any reply will be discarded. An error is reported if the reply does not arrive in five seconds (see "recall"). In normal cases, the return value is the link enclosed in the return message; it is -1 if there isn't any enclosure. Ignoring errors, the user may consider this routine an abbreviation for:

```
    struct urmesg urmess;
    struct usmesg usmess;
    usmess.usbody = outmess;
    send(ulink,link(0,CHAN16,REPLY),&usmess,NODUP);
    urmess.urbody = inmess;
    receive(CHAN16,&urmess,5);
    return(urmess.urlnenc);
```

Returned values: Under normal circumstances, the return value is either -1 or a link number. -2 means an error occurred while sending, -3 means the waiting time expired, -4 means that the return link was destroyed, -5 means that something was received with the wrong code, -6 means that a return link couldn't be created in the first place.

NOTE: CHAN16 is implicitly used; for this reason, the user is advised to avoid this channel entirely. Several other library routines also invoke "call", and thus use CHAN16.

NOTE: "Call" is not re-entrant, and so programs that use it cannot be "SHARED" (see "fork").

3.3 Close (Library Routine)

int close(file)

The argument "file" is either a link to an open file, or a terminal input or output link. The returned value is 0 on success, negative on failure (specifically, "close" is synonymous with "destroy"). These links are automatically closed when a process dies; however, execution of this command gives the process more room in its link table. Also, closing the teletype input makes it possible for another process to open it.

3.4 Copy (Library Routine)

copy(to,from) char *to,*from;

A string of length MSLEN is copied from "from" to "to". If "from" is 0, then MSLEN nulls are copied instead.

3.5 Create (Library Routine)

int create(fslink,fname,mode) char *fname;

If the file named "fname" exists, it is opened for writing and truncated to zero length. If it doesn't exist, it is created and opened for writing. The argument "fslink" is the process's link to the File System. The protection bits for the new file are specified by "mode"; these bits have the same meaning as for

UNIX files, but all files on Roscoe have the same owner. The returned values are as in "open".

3.6 Date (Service Call)

long date();

This service call returns the value of the wall clock, which is a long integer representing the number of seconds since midnight, Jan 1, 1973, CDT.

3.7 Datetol (Library Routine)

long datetol(s) char s[12];

This library routine converts a character array with format "yymmddhhmmss" into a long integer, representing the number of seconds since midnight (00:00:00) Jan 1, 1973. It accepts dates up to 991231235959 (end of 1999); -1 is returned on error.

3.8 Destroy (Service Call)

int destroy(ulink)

Link number "ulink" is removed from the caller's link table.

Returned values: 0 is returned on success. -1 means that the link number is out of range, -2 means that it has an invalid destination.

3.9 Die (Service Call)

```
die()
```

This call terminates the calling process. All links held by the calling process are destroyed.

3.10 Fork (Library Routine)

```
int fork(fname,arg,mode) char *fname;
```

The Resource Manager starts a new process running the program found in the file named "fname", which must be in executable load format. The function named "main" is called with the integer argument "arg". "Mode" is a combination (logical "or") of the following flags, defined in "user.h":

one of these:	FOREGROUND, BACKGROUND, or DETACHED
and one of these:	SHARE, REUSE, or VIRGIN

If FOREGROUND is specified, then the new process can be killed by entering a control-C on the console. FOREGROUND is mainly used by the Command Interpreter. If BACKGROUND is specified, then a "process identifier" is returned that may be used to subsequently "killoff" the child. DETACHED (i.e., neither FOREGROUND nor BACKGROUND) is the default. If SHARE is specified, then the Resource Manager will be willing to start this new process in the same code space as another process executing the same file, if that process was also spawned in SHARE mode. If REUSE is specified, the code space of an earlier process can be reused. VIRGIN means that a new copy must be loaded, and is the default. If the call succeeds, a link of type REQUEST and TELLDEST is given to the Resource Manager; the child may obtain this link by invoking "parline". The caller may receive messages from the child over this link, which has code 0 and channel CHAN14.

A returned value of -1 indicates an error. Success is indicated by a return value of 0, except in the case of BACKGROUND mode, when the return value is a "process identifier".

3.11 Fsline (Library Routine)

```
int fsline();
```

This routine returns the number of a REQUEST link to be used for communication with the File System Process. An error gives a returned value of -1.

3.12 Handler (Service Call)

```
handler(vector,func,chan) (*func)();
```

The address of a device vector in low core is specified by "vector". The interrupt vector is initialized so that when an interrupt occurs, the specified routine "func" is called at interrupt level. If the interrupt level routine performs an "awaken" call, a message will arrive on channel "chan" with urcode 0 and urnote "INTERRUPT" (see "receive").

Returned values: Success returns a value of 0. -1 means that there have been too many handler calls on that machine. -2 means that the channel is invalid. -3 means that the vector address is unreasonable. -4 means that the vector is already in use.

3.13 Inline (Library Routine)

```
int inline();
```

This routine returns the number of a REQUEST link to be used for subsequent terminal input. The Teletype Driver only allows one input link to be open at any time. An error returns a value of -1.

3.14 Kill (Service Call)

```
kill(lifeline);
```

The process indicated by "lifeline" (the return value of a successful "startup" call) is terminated. The lifeline is not destroyed.

Returned values: Success returns a value of 0. -1 indicates that the link is invalid or not a "lifeline".

Only the Resource Manager and Teletype Driver should use this call.

3.15 Killoff (Library Routine)

```
int killoff(procid);
```

This routine asks the Resource Manager to kill a process that the calling process previously created as a BACKGROUND process with a "fork" request. The value returned from that "fork" is "procid". The effect on the dead process is as if it had called "die".

0 is returned for success, -1 for failure.

3.16 Link (Service Call)

```
int link(code,chan,restr)
```

A new link is created. The calling process becomes the new link's owner (forever) and holder (usually not for very long). The caller specifies an integer, "code", which is later useful to the caller to associate incoming messages with that link. The caller also specifies "chan" as one of sixteen possibilities, CHAN1, ..., CHAN16, which are integers containing exactly one non-zero bit. Channels are used to receive messages selectively. CHAN16 should be avoided, for reasons explained in "call". CHAN15 should also be avoided, since the kernel uses it for remote loading. The returned value is the link number that the calling process should use to refer to the link. The argument "restr" is the sum of various restriction bits that tell what kind of link it is. The possibilities are:

```
GIVEALL
DUPALL
TELLGIVE
TELLDUP
TELLDEST
REQUEST
REPLY
```

"GIVEALL" means that any holder may give the link to someone else. "DUPALL" means that any holder may duplicate it (i.e., give it to someone with "dup" = DUP; see "send"). "TELLGIVE", "TELLDUP", and/or "TELLDEST" cause the owner to be notified whenever a holder gives away, duplicates, and/or destroys the link, respectively (see "receive"). A process may duplicate, give away, or destroy a newly created link without restriction and without generating notifications; restrictions and notifications only apply to links received in messages. A link must be either of type "REQUEST" or "REPLY". A REPLY link cannot be duplicated and disappears after one use; a REQUEST link can be used repeatedly unless it is destroyed by its holder. An enclosed link must always be of the opposite type from the link over which it is being sent.

Returned values: The normal return value is a non-negative link number. -1 means that the link was specified as either both or neither of REPLY and REQUEST; -2 means that the channel is invalid.

3.17 Load (Service Call)

```
int load(prog,fd,plink,arg) char *prog;
```

This call loads a program. If "fd" is -1, the console operator is requested to load "prog" manually. If "fd" is a valid link number (it should be a link to an open file) and "prog" is -1, the file is loaded on the same machine. In either of these cases, the return value is an "image", to be used for subsequent "startup" or "remove" calls.

If "fd" is a link and "prog" is a machine number, the file is loaded remotely on the corresponding machine and started. The arguments "plink" and "arg" have the same meaning as in the "startup" call. The "plink" is automatically given (not duplicated). The return value is a "lifeline", as for a "startup" call.

Returned values: 0 is returned on success. -2 and -3 mean that the link "fd" was out of range or had an invalid destination, respectively. -5 means that there wasn't room for the new image. -6 means that there are too many images. -10 means that the caller had no room for the lifeline. -11 means that the "plink" was out of range or had an invalid destination.

Only the Resource Manager should use this call.

3.18 Ltodate (Library Routine)

```
ltodate(n,s) long n; char s[30];
```

This library routine converts a long integer, representing the number of seconds since Jan 1, 1973, into a readable character string telling the time, day of the week, and date. Dates later than 1999 are not converted correctly.

3.19 Nice (Service Call)

```
nice()
```

This call allows the Roscoe scheduler to run any other runnable process. (Roscoe has a round-robin non-pre-emptive scheduling discipline; "nice" puts the currently running process at the bottom.) It is used to avoid busy waits.

3.20 Open (Library Routine)

```
int open(fslink,fname,mode) char *fname;
```

The file named "fname" is opened for reading if "mode" is 0, for writing if "mode" is 1, and for both if "mode" is 2. The argument "fslink" is the caller's link to the File System. The returned value is a link number, used for subsequent "read", "write", and "close" operations. This link may be given to other processes, but not duplicated. -1 is returned on error.

3.21 <u>Outline</u> (<u>Library</u> <u>Routine</u>)

```
int outline();
```

This routine returns the number of a link to be used for subsequent terminal output. An error returns a value of -1.

3.22 <u>Parline</u> (<u>Library</u> <u>Routine</u>)

```
parline();
```

This routine asks the Resource Manager for a link to the parent of the caller. It assumes that the parent gave the Resource Manager a REQUEST link when it spawned the child. An error returns a value of -1.

This call is typically used by a program being run by the Command Interpreter; the parent link (to the Command Interpreter) is used to get the command line arguments.

3.23 <u>Print</u> (<u>Library</u> <u>Routine</u>)

```
int print(file,format,args...) char *format;
```

This routine implements a simplified version of UNIX's "printf". The argument "file" is either a link to an open file or a terminal output link. The input is formatted and then "write" is called. The "format" is a character string to be written, except that two-byte sequences beginning with "%" are treated specially. "%d", "%o", "%c", "%w", and "%s" stand for decimal, octal, character, long integer, and string format, respectively. As these codes are encountered in the format, successive "args" are written in the indicated manner. (Unlike "printf", there are no field widths.) A "%" followed by any character other than the above possibilities disappears, so "%%" is written out as "%".

3.24 <u>Read</u> (<u>Library</u> <u>Routine</u>)

```
int read(file,buf,size) char *buf;
```

The argument "file" is either a link to an open file or a terminal input link. At most "size" bytes are read into the buffer "buf"; fewer are read if end-of-file occurs. For the terminal, control-D is interpreted as end-of-file. The returned value is the number of bytes actually read.

3.25 <u>Readline</u> (<u>Library</u> <u>Routine</u>)

```
int readline(file,buf,size) char *buf;
```

This routine is the same as "read", except that it also stops at the end of a line. For a file a "newline" character is interpreted as end-of-line; however, "readline" is very inefficient for files. For the terminal, a "line-feed" or "carriage return" terminates a line; the last character placed in the buffer will be "newline" (octal 12). Control-D or control-W will also terminate a line, but they will not be included in the bytes read. The returned value is the number of bytes read.

3.26 Recall (Library Routine)

```
int recall(inmess) char *inmess;
```

If a previous "call" (or "recall") returned a value of -3, meaning that the message did not arrive in 5 seconds, a process can invoke the library routine "recall" to continue waiting. Only the return message buffer is specified (cf. "call").

Returned values: These are the same as for "call", except that -2 and -6 don't apply.

3.27 Receive (Service Call)

```
int receive(chans,urmess,delay)

struct urmesg {          /* for receiving messages */
    int urcode;     /* chosen by user, see "link" */
    int urnote;     /* filled in by Roscoe, see "receive" */
    int urchan;     /* chosen by user, see "link" */
    char *urbody;   /* body of incoming message */
    int urlnenc;    /* index of enclosed link */
} *urmess;
```

The calling process waits until a message arrives on one of several channels, the sum of which is specified by "chans". All other messages remain queued for later receipt. The code and channel of the link for the incoming message are returned in "urmess->urcode" and "urmess->urchan", respectively. The value of "urmess->urnote" is one of five possibilities: DUPPED, DESTROYED, GIVEN, INTERRUPT, or DATA. The first three of these mean that the link's holder has either duplicated, destroyed, or given away the link (see "send" and "link"). "INTERRUPT" is discussed under "handler". "DATA" means that the message was sent by "send". The newly assigned link number for the link enclosed with the message is reported in "urmess->urlnenc"; the calling process now holds this link). If no link was enclosed, "urmess.urlnenc" is -1. Before calling "receive", the user sets "urmess->urbody" to point to a buffer of size MSLEN into which the incoming message, if any, will be put. The caller may discard the message by setting "urmess->urbody" to zero. The argument "delay" gives the time in seconds that the calling process is willing to wait for a message on the given channels; a "delay" of 0 means that the call will return immediately if no message is already there, and a "delay" of -1 means that there is no limit on how long the calling process will wait. A process can sleep for a certain amount of time by waiting for a message that it knows won't come (e.g., on an unused channel).

Returned values: 0 is returned on success. -1 means the calling process has no room for the enclosed link (the message can be successfully received later), -2 means that the argument "urmess" was bad, -3 means that the waiting time expired.

3.28 _Remove_ (_Service_ _Call_)

remove(image)

The code segment indicated by "image", the return value of a successful "load" call, is removed. Only the process that performed a "load" is allowed to subsequently "remove" that image.

Returned values: Success returns a value of 0. -1 means that the image either doesn't exist or is in use, or that the caller didn't originally load the image.

The Resource Manager uses this call to create space for new images; no other program should use this call.

3.29 Seek (Library Routine)

```
int seek(file,offset,mode)
```

The argument "file" is a link to an open file. The current position in the file is changed as specified by the "offset" and "mode". A value for "mode" of 0, 1, or 2 refers to the beginning, the current position, or the end of the file, respectively. The "offset" is measured from the position indicated by "mode"; it is unsigned if "mode" = 0, otherwise signed. A returned value of 0 indicates success, -1 indicates failure.

3.30 Send (service call)

```
int send(ulink,elink,usmess,dup)

struct usmesg {          /* for sending messages */
    char *usbody;  /* body of message to be sent */
} *usmess;
```

This call sends a message along link number "ulink". The address of the message body, a string of MSLEN bytes, lies in "usmess->usbody". If no message is to be sent, "usmess->usbody" is zero. If the caller wishes to pass another link that it holds with the message, it specifies that link's number in "elink" (the "enclosed link"). If there is no enclosure, "elink" should be -1. The use of elinks is restricted in various ways; see "link". The argument "dup" is either "DUP" or "NODUP"; in the first case, the enclosed link is duplicated so that both the sender and receiver will hold links to the same owner; in the second case, the enclosed link is given away so that only the receiver of the message will hold it.

Returned values: 0 is returned on success. -1 means that the ulink number is bad, -2 means that the ulink's destination is not valid (the number is in the right range, but does not correspond to any active link). -3 and -4 have corresponding meanings for the elink. -5 means that the message was bad, -6 means that the elink can't be duplicated, and -7 means that the elink can't be given away.

No error is reported if the destination process has terminated; in this case, the message is discarded.

3.31 Setdate (Service Call)

```
setdate(n) long n;
```

This service call sets the wall clock to "n", which is a long integer representing the number of seconds since midnight, Jan 1, 1973.

Only the Command Interpreter and Resource Manager use this call.

3.32 Startup (Service Call)

```
int startup(image,arg,plink,dup)
```

This call starts a process whose code segment is indicated by "image", the return value of a successful "load" call. The child is given "arg" as its argument to "main". The child's link number 0 is "plink", a link owned by the caller; this link is either given to the child or duplicated depending on whether "dup" is NODUP or DUP, respectively. The child cannot destroy link 0.

Returned values: Success returns a non-negative lifeline number, which can be used for a subsequent "kill". -1 means that the caller had no room for the lifeline. -2 or -3 means that the "plink" was out of range or had an invalid destination, respectively. -4 means that there was no room for the new process's stack. -5 means that the "image" was invalid.

Only the Resource Manager should use this call.

3.33 Stat (Library Routine)

int stat(fslink,fname,statbuf) char statbuf[36];

This library routine gives information about the file named "fname". The argument "fslink" is the process's link to the File System. An error returns a value of -1. After a successful call, the contents of the 36-byte buffer "statbuf" have the following meaning:

```
struct{
    char minor;         minor device of i-node
    char major;         major device
    int  inumber;
    int  flags;
    char nlinks;        number of links to file
    char uid;           user ID of owner
    char gid;           group ID of owner
    char size0;         high byte of 24-bit size
    int  size1;         low word of 24-bit size
    int  addr[8];       block numbers or device number
    long actime;        time of last access
    long modtime;  time of last modification
    } *buf;
```

NOTE: Some of these fields are irrelevant, since all Roscoe files have the same owner.

3.34 Time (Service Call)

```
long time();
```

This service call returns a long integer that may be used for timing studies. The integer is a measure of time in intervals of ten-thousandths of seconds. NOTE: The time wraps around after a full double word (32 bits).

3.35 Unlink (Library Routine)

```
int unlink(fslink,fname) char *fname;
```

This library routine removes the file named "fname"; it cleans up after "create" and "alias". The argument "fslink" is the process's link to the File System. Errors return a value of -1.

3.36 Write (Library Routine)

```
write(file,buf,size) char *buf;
```

The argument "file" is either a link to an open file or a terminal output link. Using this link, "size" bytes are written from the buffer "buf". There are no return values.

4. CONSOLE COMMANDS

The Command Interpreter is a utility process that reads the teletype. When the Command Interpreter is awaiting a command, it types the prompt ".". A command consists of a sequence of "arguments" separated by spaces. Otherwise, spaces and tabs are ignored except when included in quotation marks ("). Within quotes, two consecutive quotes denote one quote; otherwise, quotation marks are deleted. The first "argument" is interpreted as a "command" (see below). Command names may be truncated, provided the result is unambiguous. It is intended that all commands will differ in their first three characters.

The following is an alphabetized list of console commands.

4.1 alias <filename1> <filename2>

The second indicated file becomes another name for the first indicated file. If either of these is "deleted", the other (logical) copy still exists; however, changes to either affect both.

4.2 background <filename> <arg>

The indicated file must be executable. It is started as a BACKGROUND process, with the integer argument "arg". The Command Interpreter prints out the new process's process identifier, which may be used for subsequent "killing" and then gives the next prompt.

4.3 _copy_ <_filename1_> <_filename2_>

The second indicated file is created with a copy of the contents of the first indicated file.

4.4 _delete_ <_filename_>

The indicated file is deleted.

4.5 _directory_ <_filename_>

Status information for the indicated file is typed.

4.6 _help_

A list of available commands is displayed.

4.7 _kill_ <_arg_>

The indicated argument should be the process identifier returned from a previous "background" command. The process referred to by the process identifier is killed.

4.8 _make_ <_filename_>

The named file is created. Subsequent input is inserted into the file; the input is terminated by a control-D.

4.9 run <filename> <arg1> <arg2> ...

The indicated file should be an executable file. It is run as a FOREGROUND process. The Resource Manager is given a REQUEST link, which the new process may use to ask for the command line arguments. When the loaded program starts up, the argument to "main" tells the number of command line arguments. To get the individual arguments, the loaded program sends a message to the Command Interpreter (its parent). The first word of the message is ARGREQ, and the second is an integer specifying which argument is desired. The name of the program is argument number 0. The returned message body is the argument, which is a null-terminated string of length at most MSLEN.

4.10 set <modelist> or SET <modelist>

This command changes the console input modes. The mode list is a sequence of keywords "x" or "-x", where "x" can be any of the following:

upper	(the terminal is upper case)
echo	(the terminal echoes input)
hard	(the terminal is hard-copy)
tabs	(the terminal has hardware tabs)

Keywords may be abbreviated according to the same rules as commands. The format "x" turns on the corresponding mode, "-x" turns it off. (UPPER is recognized for upper; "lower" means "-upper".) For more information, see the section "CONSOLE INPUT PROTOCOLS".

4.11 <u>time</u> <<u>format</u>>

If a format is given (as "yymmddhhmm"), the wall clock is set to that time, and printed. With no argument, "time" prints the wall clock time.

4.12 <u>type</u> <<u>filename</u>>

The indicated file is typed.

5. CONSOLE INPUT PROTOCOLS

The Teletype Driver performs interrupt-driven I/O, which allows for typing ahead. Also, the following characters have special meanings:

Control-C	kill the running program (but don't kill the command interpreter itself)
Control-D	end of file (terminates a "read" or "readline")
Control-W	end of line (but no character sent)
line-feed	end of line
carriage return	end of line
rubout	erase last character (unless line empty)
Control-X	erase current line
escape	next character should be sent as is

In "echo" mode, input is echoed, otherwise not. In "hard" mode, output is designed to be legible on hard copy devices; otherwise the Teletype Driver assumes that the cursor can move backward, as on a CRT. In "tabs" mode, advantage is taken of hardware tabs on the terminal. In "upper" mode, the terminal is assumed to only have upper case. Input is converted to lower case, unless escaped. Upper case characters are printed and echoed with a preceding "!". Escaped [,], @, ^, and \ are converted to {, }, `, ~, and ¦, respectively, and the latter are similarly indicated by preceding "!"s.

6. UTILITY PROCESS PROTOCOLS

This section describes the protocols that user programs must follow to communicate with the utility processes when the library routines described earlier are inadequate. Four utility processes are the Resource Manager, the File System Process, the Teletype Driver, and the Command Interpreter. The Resource Manager keeps track of which programs are loaded and/or running on the local machine. The kernel and the Resource Manager reside on each machine. The Teletype Driver governs I/O on the console; the Command Interpreter interprets console input. The File System Process implements a file system by communicating with the PDP-11/40. It need not exist on every machine.

During Roscoe initialization, one Resource Manager is started. It loads a full complement of utility processes (the Teletype Driver, Command Interpreter, and File System Process) on its machine and various utility processes on the other machines. When a particular Resource Manager is not given a local Teletype Driver or File System Process, it shares the one on the initial machine.

6.1 Input/Output Protocols

This section describes the message formats used for communicating with the File System and Teletype Driver Processes. A program that explicitly communicates with the File System Process or Teletype Driver must include the header files "filesys.h" and "ttdriver.h", which define the necessary structures.

To open an input or output line to the terminal, to change the modes on the terminal, or to inform the teletype of whom it should kill when encountering a control-C, a message is sent over the terminal link of the following form:

```
struct ttinline{
     char tticom;
     char ttisubcom;
     char ttimodes;
}
```

"tticom" is either OPEN, STTY, MODES, or TOKILL. In the case of OPEN, "ttisubcom" is either READ or WRITE, and the return message has the new link enclosed In the case of STTY, "ttimodes" tells what the new modes should be (a bit-wise sum of ECHO, TABS, HARD, and UPPER). In the case of MODES (to find out the current modes), the return message has the modes in "ttimodes". In the case of TOKILL (to inform the Teletype Driver which process to kill on receipt of control-C), the message encloses a lifeline.

To open, create, unlink, alias, or get status information on a file, a message is sent over the file system link in the following form:

```
struct ocmesg{
     int ocaction;
     int oclength;
     int ocmode;
}
```

"ocaction" is either OPEN, CREATE, UNLINK, ALIAS, or STAT. "oclength" tells the length of the file name; in the case of ALIAS, this field contains the concatenation of two file names. "ocmode" is the mode for OPEN or CREATE; in the case of ALIAS, it holds the length of the first file name. The file system sends back a message with an enclosed link, over which the file name is sent. This message again has an enclosed link for the File System Process's next response. In the cases of OPEN or CREATE, a successful return contains a valid enclosed link; for UNLINK, STAT, or ALIAS, there is no enclosed link. In the case of STAT, the return message has the structure of a "rdmesg" as in the case of READ below; the first word is 36 for success, -1 for failure, and the next 36 bytes of the message are the result of the stat. In all other cases, the first word is 0 on success, -1 on failure.

For either the terminal or the file system, reading or writing is done by sending a message of the following form:

```
    struct fsmesg{
         int fsaction;
         int fslength;
         char fstext[MSLEN-4];
    }
```

"fsaction" should be either READ, READLINE, or WRITE. "fslength" tells how many bytes are intended to be read, or are being sent to be written. In the case of WRITE, the text is sent in subsequent messages, and nothing is returned. In the cases of READ or READLINE, the response is of the following form:

```
    struct rdmesg{
         int rdlength;  /* amount actually read */
         char rdtext[MSLEN-2];
    }
```

The maximum allowable read is size MSLEN-2.

To perform a seek on an open file, send a message to the file system of the following form:

```
    struct skmesg{
         int skaction;  /* should be SEEK */
         int skoffset;
         int skmode;
    }
```

Any enclosed link in the return message indicates success, and should be immediately destroyed.

6.2 Resource Manager Protocols

Processes that communicate explicitly with the Resource Manager must include the header file "resource.h". The following structure is declared there:

```
struct rmmesg {     /* messages to Resource Managers */
    int rmreq;      /* type of request */
    int rmarg;      /* various miscellaneous arguments */
    int rmmode;     /* the mode for STARTs or KILLs */
}
```

The Resource Manager keeps track of which images (code segments) and processes exist. A separate Resource Manager runs on each machine in the network; these programs communicate with each other, but are relatively independent.

Each Resource Manager holds a terminal link and file system link, which are either for local utility processes or else links received from the first Resource Manager initialized. Whenever a Resource Manager has a local terminal it also has a local command interpreter.

There are three kinds of processes: FOREGROUND, BACKGROUND, and DETACHED. When a process is started, its link 0 is owned by the local Resource Manager, to whom all of this process's requests are directed.

The first FOREGROUND process for any terminal is always the

Command Interpreter, which initially "has the ball". Each terminal always has one FOREGROUND process that "has the ball". The process "with the ball" may create another FOREGROUND process, which means that the child now "has the ball". The meaning of "having the ball" is that a control-C entered on the corresponding terminal will terminate the process. When the process "with the ball" terminates, its parent then "recovers the ball", and will be terminated by the next control-C. If one of the processes in this FOREGROUND chain terminates, the chain is relinked appropriately. The command interpreter is an exception in that control-C's have no effect on it.

A process may also create another process as a BACKGROUND process. In this case, the child's process identifier is returned to the parent, and later the parent can use this identifier to terminate the child. These identifiers are assigned by the Resource Manager, and are distinct from the process identifiers used in the kernel.

A DETACHED process cannot be terminated by either method.

A user may make five kinds of requests on its Resource Manager:

1. RMTTREQ Request

The Resource Manager is requested to give the requestor a link to the requestor's terminal. This link will be sent over the enclosed link in the request, which should therefore be a REPLY link.

2. RMFSREQ Request

The Resource Manager duplicates its file system link and sends it back over the enclosed link in the request, which should therefore be a REPLY link.

3. RMSTART Request

The Resource Manager will start a process, using the link enclosed with this request for two purposes: 1) to respond to the request (see conditions for response below), or 2) to save it and give to the child if the child asks for it (see RMPLINK below). The caller must be careful, of course, not to give a REPLY link if both uses are intended. Also, the caller must make the enclosed link GIVEALL if the Resource Manager should try to load the process on another machine, rather than giving up if it doesn't fit on the local one. The RMSTART request also specifies the file name and an integer argument to be given to the child when it starts.

The caller also specifies a "mode" for starting the child, which is a combination of bits with various meanings. The user should specify either BACKGROUND, FOREGROUND, or DETACHED (the default is DETACHED). FOREGROUND is only allowed if the requestor currently "has the ball" for its terminal. The user should specify either SHARE, REUSE, or VIRGIN (the default is VIRGIN). These alternatives are described above (see "fork"). The user should also specify either GENTLY or ROUGHLY (the default is GENTLY). If GENTLY, the Resource Manager will first try to load it locally without throwing out any other unused images, and then will try to do the same on other machines. When this fails, or if ROUGHLY was specified, it tries to make room locally for the new process, and then tries to do so on other machines. The user should also specify either ANSWER or NOANSWER (the default is NOANSWER). If ANSWER is specified, or if BACKGROUND was specified, then the Resource Manager sends a reply over the enclosed link. The first word of the reply is the return code; -1 always means failure; 0 means success except in the case of BACKGROUND, when the value returned is the process identifier of the child.

An existing code segment is reusable if the filename still refers to an existing publicly executable load format file that has not been modified since the copy in question was loaded. Any number of processes may share a code segment. The terminal associated with a child process is always the same as the one associated with its parent; the command interpreter is loaded with a terminal during initialization.

4. RMKILL Request

The Resource Manager kills the process whose process identifier is given as part of the request. The request may enclose a link that is used to give a one-word acknowledgement of success or failure if the request specifies ANSWER (as in RMSTART, described above). The process being killed must of course be BACKGROUND, and only the process that started it is allowed to kill it.

5. RMPLINK Request

The Resource Manager returns the link that was originally enclosed with the request that started this process. It is returned over the link enclosed with the RMPLINK request, which must therefore be of the proper type, whichever that may be.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the assistance of the following graduate students who have been involved in the Roscoe project: Jonathan Dreyer, Jack Fishburn, Michael Horowitz, Will Leland, Paul Pierce, and Milo Velimirovic. Their hard work has helped Roscoe to reach its current level of development and will be essential in completing its design and implementation.

REFERENCES

Baskett, F., Howard, J. H., Montague J. T.,
"Task Communication in Demos",
Proceedings of the Sixth Symposium on Operating Systems Principles,
pp. 23-31,
November 1977.

Finkel, R. A., Solomon, M. H.,
Processor Interconnection Strategies,
University of Wisconsin -- Madison Computer Sciences
Technical Report #301,
July 1977.

Finkel, R. A., Solomon, M. H.,
The Roscoe Kernel,
University of Wisconsin -- Madison Computer Sciences Technical Report #337,
September 1978.

Ritchie, D. M.,
C Reference Manual,
Unpublished memorandum, Bell Telephone Laboratories,
1973.

Ritchie, D. M., Thompson, K.,
"The UNIX Time-Sharing System",
Communications of the ACM,
Vol. 17, No 7,
pp. 365-375,
July 1974.

Solomon, M. H., Finkel, R. A.,
ROSCOE -- a multiminicomputer operating system,
University of Wisconsin -- Madison Computer Sciences
Technical Report #321,
September 1978.

Tischler, R. L., Finkel, R. A., Solomon, M. H.,
Roscoe Utility Processes,
University of Wisconsin -- Madison Computer Sciences
Technical Report #338,
September 1978.