ROSCOE UTILITY PROCESSES

by

Ron Tischler
Raphael Finkel
Marvin Solomon

Computer Sciences Technical Report #338

February 1979

ROSCOE UTILITY PROCESSES*

February 1979

Ron Tischler
Raphael Finkel
Marvin Solomon

Technical Report 338

Abstract

Roscoe is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. Roscoe consists of a kernel program resident on each computer and several utility processes. This document describes the implementation of the Roscoe utility processes at the level of detail necessary for a programmer who intends to add a module or modify the existing code. Companion reports describe the purposes and concepts underlying the Roscoe project, present the implementation details of the kernel, and display Roscoe from the point of view of the user program.

---

*This research was supported in part by the United States Army under contract #DAAG29-75-C-0024.

TABLE OF CONTENTS

ROSCOE UTILITY PROCESSES

This paper documents the source code for the following Roscoe utilities:

resource manager
terminal driver
command interpreter
file manager
the "demon" (a PDP-11/40 process with which the file manager communicates)
user-callable library routines
copyfile (a program used implicitly by the command interpreter)

The reader is assumed to be familiar with the Roscoe User Guide [Tischler, Solomon, and Finkel 78], which describes the purposes and use of these utilities. The present paper consists of a detailed explanation of the programs and data structures for those who intend to help maintain these utilities. The Roscoe kernel code is similarly documented [Finkel and Solomon 78].

The documentation given here is accurate as of January 20, 1979. However, recent developments will soon cause some modifications to the processes discussed here. In particular, a new utility process called a "pipe" has been introduced to attach the output of one process to the input of a second one. The command interpreter and the resource manager will cooperate to establish piped processes.

Unless otherwise stated, all files mentioned are in the directory "/usr/network/roscoe/user".

1. THE RESOURCE MANAGER

1.1 General

The code lies in "resource.u". Programs that communicate with the resource manager should include "resource.h" unless all such communication is handled by library routines.

The resource manager may use the service calls "load", "remove", "startup", and "kill". These calls are meant to be privileged, although that restriction is not yet enforced. The "startup" call gives the new process a link to its resource manager. This link should be of a special kind, although the resource manager currently refers to it as a "REQUEST" link. Either a REQUEST or REPLY link may be enclosed over this link. Currently, the kernel does not enforce the restrictions on enclosed links. Furthermore, the new process may not destroy this parent link except by dying. This restriction is enforced.

1.2 Protocols between resource managers

When resource managers talk to each other, they send requests whose "rmreq" fields hold special values. These values are defined by macros that begin with the letters "RR". We will follow the convention of calling the originator of such a message the "first" resource manager and the recipient the "second". Together, they are called "colleagues". When client processes talk to resource managers, they send requests whose "rmreq" fields are

defined by macros that begin with the letters "RM". Following sections describe how these codes are employed to carry out the various resource manager functions.

The routine "sendrm" is used to send messages between resource managers. The array "rmtab", of size 5, keeps track of which other resource managers exist. Entries in "rmtab" are link numbers; -1 indicates that there is no corresponding resource manager. Links used between resource managers use channel 2, and the code is always the machine number of the holder.

Resource managers may make RMFSREQ or RMTTREQ requests of each other, in which case the request is treated the same as any other user's request, except that for RMTTREQ, the local terminal link is assumed to be the one desired. In addition, there are five other requests peculiar to resource managers, as listed below. Whenever these requests are made, the first resource manager does not wait for a reply; any reply that eventually comes will be self-explanatory.

RRSTART: This request continues an RMSTART request that the first resource manager could not complete. Everything in the original request is passed along; no response is needed. The resource managers pass the request around in order of increasing machine id. The originator recognizes it should it return. This circular method is an ad-hoc approach that will be changed in the future to a more reasonable polling order.

RRKILL: This request continues an RMKILL request if the process targeted for the kill is not local to the first resource manager. The request is forwarded to the resource manager on the

proper machine; no response is needed.

RRLINK: This message asks the second resource manager for a link owned by that second resource manager. The first resource manager intends to give this link to a third resource manager.

RRINFORM: This message accompanies an enclosed link owned or held by the first resource manager. (See Section 1.3.)

RRPASS: Used to "pass the ball" when a FOREGROUND process "with the ball" for a certain terminal has died, and the process that should next "get the ball" is on another machine. (See Section 1.8).

1.3 Resource manager initialization

When a resource manager is loaded by the kernel job of the Roscoe kernel, it receives the machine number as the argument to "main". In particular, if the bit "NOTPAPA" is off, this resource manager knows that it is the first one. We will call such a resource manager "original". A resource manager that the kernel job starts in an attempt to recover from failure at some node or as a subsidiary resource manager has the bit NOTPAPA set.

When the original resource manager starts, initialization is done by "initrm0". A file manager and terminal driver are loaded and started as DETACHED processes; the file manager is loaded manually, and does not occupy a spot in "imagetab". Input and output terminal links are opened, a "configuration" is read from the terminal by "readline", and the input link is closed. The "configuration" is a character string that the resource manager scans to determine what other resource managers to load and with

what arguments. For example, the configuration

1T24FT

indicates that resource managers should be loaded on machines 1, 2, and 4; machine 1 will have its own terminal, machine 4 will have its own terminal and file system, and machine 2 will have neither.

The argument given to a remote resource manager has the machine number as its lowest three bits and contains flags RMTTFLAG and RMFSFLAG to indicate respectively whether a terminal driver (and attendant command interpreter) and file manager should be loaded locally. Also, the bit "NOTPAPA" is set to indicate that the child resource manager is not the first one started. The high order byte of the argument gives the machine number of the parent (papa).

When a resource manager other than the original one starts, it uses the initialization routine "initrms". An entry is made in "rmtab" for the first resource manager (the owner of this resource manager's link 0), and an RRINFORM message is sent to that resource manager with an enclosed link having channel 2. The code for this link is the number of the first resource manager. The "rmarg" field of this initial message is -1. (The discussion of RRINFORM messages continues below.) If the RMFSFLAG is on, a local file manager is loaded by the routine "loadfs", which asks the first resource manager for a file system link, uses it to perform the load, and then destroys this unneeded link. If the RMTTFLAG is on, a local terminal driver and command interpreter are loaded. If these flags are off, the ap-

propriate links are obtained from the first resource manager by the same RMTTREQ and RMFSREQ protocols followed by any other process.

The routines "loadtt" and "loadci" are used to load local copies of the terminal driver and command interpreter, respectively. No matter how a terminal is obtained, a terminal output link is automatically opened. The variable "owntt" tells which terminal (0-4) the resource manager is using. The command interpreter is vaccinated against control-C's by setting its "lifeno" field in "proctab" to -1.

The routine "rrinform" handles RRINFORM requests. If the "rmarg" field is -1, the receiver knows that a new resource manager just came to life. The number of this new resource manager can be found in inmess.urcode. The receiver then acts as the "papa" and sends out RRLINK messages to begin the process of hooking together all the other resource managers. Otherwise, the high order byte of the "rmarg" field tells the number of the resource manager that owns the link, and the low order byte tells for which resource manager the link is intended. If this intended holder is not the present resource manager, the RRINFORM request is forwarded to the correct one. Whenever a resource manager receives a link which it will continue to hold, it updates its "rmtab" accordingly.

The routine "rrlink" handles RRLINK requests which, as mentioned above, are only sent by the original resource manager to other resource managers. A link is created on channel 2; the code is specified by "rmarg", which indicates the resource

manager that will eventually hold the link. The link is sent to the first resource manager in an RRINFORM message; it will then forward it to the intended holder, as described above.

1.4 Wall clock synchronization

When the original resource manager starts, a special request is sent to the demon on the PDP-11/40 for the Unix date. The variable "timewarp" is used to convert between Roscoe time and Unix time (the former begins Jan 1 1973 CST; the latter Jan 1 1970 GMT). Other resource managers initialize their dates to zero, but this value is soon corrected.

Whenever "sendrm" is used (to send an RRSTART, RRKILL, RRINFORM, RRLINK, or RRPASS message to a colleague), the current date is placed in the "update" field of the message. Whenever such a request is received, the local date is set to the value in the "update" field if it is later. This algorithm keeps the wall clocks in the various Roscoe kernels from losing time relative to each other.

1.5 Process initiation

The resource manager knows which client sent each RMSTART request because the code of the link containing the request is also the index for that process in the resource manager's process table. The resource manager can also determine the client's associated terminal from this table. If the request cannot be processed locally (either the "load" or "startup" service call fails due to lack of room), then an RRSTART request is sent to the resource

manager with the next higher machine number (modulo 5), as determined from "rmtab". The RRSTART request has all the information of the RMSTART request (including the same enclosed link, if any), plus the client's process identifier and terminal number, which would not otherwise be known to the second resource manager. The second resource manager tries likewise to initiate the child process, and if it also fails, sends the request on further. The identifier of the child includes its machine number as its lowest three bits; if the RRSTART returns to the client's resource manager, it is recognized as a failed request. It may be sent around once more (if the original method involved the GENTLY mode) but, in any case, the buck stops somewhere, either with success or failure. A reply (if required) is sent to the client from the resource manager where the algorithm stops.

The routines "rmstart" and "rrstart" are invoked for RMSTART and RRSTART requests, respectively. Each of these routines computes the client's process identifier and terminal number in its own way and then calls "rawstart". Another argument to "rawstart" tells whether the load should have GENTLY or ROUGHLY mode. An RRSTART message received at the client's machine is recognized by "rrstart"; if the mode was GENTLY, "rawstart" is now called with ROUGHLY mode; if the mode was ROUGHLY, a negative reply is sent to the client. If the load doesn't succeed locally and there are no other machines, "rawstart" similarly calls itself with mode ROUGHLY (if the mode was GENTLY), or sends the user a negative reply (if the mode was ROUGHLY).

The routine "getimage" loads a program. A "stat" checks that

the file is a publicly executable load-format file and computes its date of last modification. If no acceptable copy already resides locally, a new one is loaded. If "imagetab" is full, an unused image is removed (in ROUGHLY mode). The procedure "makeroom" is used to remove unused images. Unused images are also removed (in ROUGHLY mode) until there is room for the new image to be loaded. Images are removed in ascending order of their index in "imagetab". When a new image is loaded, "imagetab" is updated accordingly. Future developments should prevent loading if a colleague has a useable copy, and removal of images should perhaps use some other algorithm.

The routine "newproc" starts a process. The new process is given a link to the resource manager on channel 1; this link has type REQUEST and TELLDEST, and its code is the new child's index in "proctab". If the start succeeds, the corresponding "count" field in "imagetab" is incremented, and an entry is made in "proctab". If the start fails because there was no room for the process's stack, then "makeroom" is called, as in the case of "getimage" described above. The lowest three bits of the child's process identifier tell the machine number; the variable "uniquecode" generates unique process identifiers. If the child is to be "DETACHED", its lifeline is destroyed.

1.6 Process termination

The "rmarg" field of an RMKILL request tells the process identifier of the victim. The resource manager figures out which colleage hosts the victim by looking at the lowest three bits, and then either completes the request itself or forwards an RRKILL request to the appropriate colleague. The RRKILL request contains the process identifier of the client, which otherwise wouldn't be known to the colleague and which is needed to check that the client has permission to kill the victim. The routine "rawkill" checks this permission and then performs the kill; the victim's lifeline isn't destroyed (yet). There's nothing to prevent the parent of a FOREGROUND process from performing a kill if it correctly guesses the child's process identifier.

When a client terminates, naturally or otherwise, the resource manager receives a DESTROYED message on its link and calls "procdie". The client's entry in "proctab" is deleted by setting the "proctype" field to UNUSED, except for FOREGROUND processes (see further discussion below). The corresponding "count" field in "imagetab" is decremented. The process's parent link and/or lifeline are destroyed if the resource manager still holds them.

The termination of a colleague is similarly detected. The routine "rmdie" updates "rmtab" accordingly.

1.7 Terminal links

When a request for a terminal link is received over channel 1, the corresponding "ttypeno" field in "proctab" is compared to "owntt" to see if the local terminal link is desired. If not, the routine "gettt" is used to ask the appropriate colleague for a copy of its terminal link. RMTTREQ requests received over channel 2 (i.e., from a colleague) are always given a copy of the local terminal link.

1.8 FOREGROUND processes

One linked list of FOREGROUND processes is associated with each terminal; at most one terminal is owned by each resource manager. The process that "has the ball" (will be killed by the next Control-C) is at the head of this list, and it points to the next process to "get the ball". Segments of this list reside physically on each machine; each list logically threads its way among several machines.

Two fields in a process table entry are relevant to this discussion. The field "parentno" is the process identifier of the process's parent; the last three bits of this number tell the parent's machine number. The field "parentp" is the index in "proctab" of the next local item in the FOREGROUND list. When a process's successor is on the same machine, "parentp" points to it, and the last three bits of "parentno" are the machine id; when a process's successor is on another machine, the last three bits of "parentno" tell which machine to go to next, and

"parentp" tells which local process comes next when the chain returns to this machine. Each terminal chain in each resource manager has a special header node containing two fields: "foretop" gives the index of the first item in the process table (i.e., it corresponds to a "parentp"), and "theball" is a Boolean that tells whether this machine (i.e., the process indicated by "foretop") "has the ball". A null pointer value for "parentp" or "foretop" is indicated by -1.

When a resource manager receives an RMSTART request with FOREGROUND mode, "rmstart" checks that the client "has the ball" and turns off its "theball" flag. If the load doesn't succeed locally, "rawstart" sends an RRSTART message as usual. Wherever the load succeeds, the routine "newproc" will turn on the "theball" flag, and insert the new process at the head of the appropriate list. The routine "telltt" is used by "rawstart" to send a TOKILL message to a terminal driver (local or not), so the terminal driver will know which process now "has the ball". If the start fails, the resource manager that initiated it notices the request returning (perhaps for the second time); its "theball" flag is turned back on by "rrstart", so the process that previously "had the ball" still does.

When a FOREGROUND process dies, "procdie" marks the corresponding "proctype" entry in "proctab" as DEFUNCT, rather than UNUSED. The "parentno" and "parentp" fields are still relevant, so the item is still linked up. These DEFUNCT items are cleaned off as FOREGROUND processes die in the "proper" sequence. Specifically, when the process "with the ball" dies,

"procdie" turns off the "theball" flag and calls "rrpass" to clean off DEFUNCT processes at the head of the FOREGROUND list so long as they point to other processes on the same machine. If a non-DEFUNCT process is reached in this manner, it then "has the ball"; the "theball" flag is turned on, and "telltt" is used to send an appropriate TOKILL message to a terminal driver. On the other hand, if the list points to another machine, an RRPASS message is sent to the appropriate colleague. When a resource manager receives an RRPASS message, it also uses "rrpass" to clean off DEFUNCT processes and/or "pass the ball".

Files

resource.u, resource.h

Data Structures

```
struct rmmesg {     /* messages to resource managers */
      int rmreq;    /* type of request */
      int rmarg;    /* various miscellaneous arguments */
      int rmmode;   /* the mode for STARTs or KILLs */
      long update; /* time field used between R.M.'s */
      int parno;    /* parent's proc. id., used by R.M.'s */
      int ttno;          /* a terminal number, used by R.M.'s */
      } rmmess;  /* contents of outmess, used implicitly */
struct { /* image table entry */
      char fname[RMFNAMESZ]; /* file name */
      long loadtime;          /* time it was loaded */
      int count;         /* number of active processes */
      int procmode;           /* SHARE, REUSE, or VIRGIN */
      int imageno;       /* image, used for "start" or "remove" */
      } imagetab[NBRIMAGES]; /* image table */
struct procnode {  /* process table entry */
      int  proctype;      /*  FOREGROUND,  BACKGROUND,   DETACHED,
     UNUSED, or DEFUNCT */
      int parentp; /* an index in this table */
      int parentno;      /* process identifier of the parent */
      int plink;    /* link supplied by parent during start */
      int location;      /* index into the image table */
      int lifeno;  /* lifeline, used for "kill" */
      int procid;  /* its process identifier */
      int ttypeno; /* its terminal number */
      } proctab[NBRPROCS];  /* known process table */
```

Procedures

main(arg)

Initializes the resource manager for machine given in "arg",

executes a loop that receives and dispatches requests. The argument also contains the bit NOTPAPA.

respond(n)
Sends a one-word response to the client. If it is a negative error indicator, destroys the link that client submitted.

giveaway(elink,how)
Returns the "elink" to the current client. "How" is either "DUP" or "NODUP" to govern the disposition of that link.

int sendrm(n,elink,how)
Sends a message to another resource manager. "n" is the index of the resource manager to send it to, "elink", "how" describe the link to enclose; "elink" = NOLINK means to really not send a link. Returns 0 on success, -1 on failure; sometimes the caller cares.

cryout(message) char *message;
General-purpose error indication routine.

gettt(n)
Asks the resource manager on machine n for its terminal link. Returns either the link or -1 for error.

telltt(tt,elink)
Tells terminal on link "tt" that its new killlink is "elink".

loadfs()
Gets a file manager link from the original resource manager and uses it to load a file manager on this machine.

loadtt()
Loads a terminal driver on this machine and gets an output link to it.

loadci()
Loads a command interpreter on this machine; assumes there is already a terminal driver.

initrm0()
Initialization specific to the original resource manager. Finds the local time from Unix, loads the first file manager via a manual load, loads a terminal driver, gets an input line to ask for the configuration, then decodes the configuration and loads the other machines.

initrms(arg)
Initialization specific to non-original resource managers. Informs the original resource manager; either loads a local file manager and terminal or uses links to the original resource manager's copies.

rmstart()
Handles client request to start a new process. If FOREGROUND, insures the client currently has the ball. Calls "rawstart".

rrstart()
Handles request from another resource manager to do a start. If the request has come full circle, either gives up or tries roughly. Calls "rawstart".

rawstart(parent,ttype,how)
Tries to start a process on this machine. "Parent" gives the procid of the client who initiated it all, "ttype" gives

its teletype number, "how" is GENTLY or ROUGHLY, but matters only inside "getimage". Calls "newproc".

rmkill()
Handles client request to kill a process. Either directs the request to the appropriate resource manager or calls "rawkill".

rawkill(parent)
Checks if the parent has the right to submit this kill request; if so, submits a "kill" service call.

rrlink()
The papa resource manager has requested a link for a third party. The third party's number is in "contents->rmarg"; the papa's number is "inmess.urcode". Prepares a link and sends it.

rrinform()
Handles a new link given by a colleague. Establishes knowledge of that colleague in the proper tables. During recovery actions, the new information may disagree with the old.

rrpass()
This resource manager has just been given the ball. Cleans off the defunct part of the foreground stack, and if it becomes empty, sends the ball elsewhere.

rmdie()
Just found out that a colleague has died. Clears out entries in "rmtab".

int getimage(name,mode,how) char *name;
Loads a new core image, and returns an index into imagetab. If "how" = ROUGHLY, will also try to make room; otherwise just hopes there is room, or a usable copy exists. The mode is SHARE, REUSE, or VIRGIN. Returns -1 if the load fails. Checks that the image is executable, and will not use an existing image with obsolete date. Discovers if the image is an Elmer program.

int newproc(imagno,arg,parent,type,ttype)
Makes a new process by using the service call "startup". Returns an index in the updated proctab or -1 for error. "Imagno" is the index in imagetab for the process's image, "arg" is the argument to give the new process, "parent" is the parent's procid, "type" is BACKGROUND, FOREGROUND, or DETACHED, "ttype" tells which terminal to use. If the process is in Elmer, opens its object file in order to give the link to "startup".

procdie()
Cleans up after the termination of a client. If it was in the foreground and had the ball, the ball is passed, possibly to a colleague.

int getindex(i)
Returns the index in proctab for the process whose procid is i.

int makeroom()
Makes room in the image table if possible, and returns the index of an available slot. If necessary, core images of terminated processes are removed. Returns -1 on failure.

drwrite(word)
Busy-waits until the DR-11 line to Unix is ready, then writes one word.

int drread()
Busy-waits until the DR-11 line from Unix is ready, then returns one word read.

2. THE TERMINAL DRIVER

2.1 General

The code lies in "ttdriver.u". Processes using the terminal driver should include "ttdriver.h" and "filesys.h". (The latter contains macros used by both the terminal driver and file manager.) It isn't necessary to include these if all communication is done by library routines.

The terminal driver gives its parent (usually a resource manager) a REQUEST link on channel 1. All requests to open the terminal for input or output come over this link or copies thereof; also, the resource manager sends "TOKILL" messages over this link. (See Section 2.8.) An "input link" is used for "read" and "readline" requests from the client; an "output link" is used for "write" requests. Thus, the terminal driver accepts data on its output links and supplies data on its input link. At most one input link and NUMCODES (currently 5) output links may be open. (Channel 2 is used for output, channel 3 for input.) These links are of type GIVEALL but not DUPALL. Destruction of such a link is interpreted as a "close" command. The holder of the input link may also use it to request or change terminal modes. (See Section 2.4.)

Input and output are interrupt-driven; the routines "ttindriv" and "ttoutdriv" are invoked at interrupt level when the corresponding devices become ready. Initially, interrupts are enabled and "handler" calls set up the interrupt vectors. The interrupt-level routines send "awaken" messages to the terminal driver on channel 10. Interrupts are disabled at crucial times by turning off the appropriate interrupt-enable bits in the device registers. (Since interrupt-level routines run at high priority, this interrupt disabling is not strictly necessary.) These interrupt-driven routines share data with the rest of the terminal driver.

2.2 Overview of input

The Boolean variable "inuse" tells whether an input link is open. The routine "readmsg" executes a "read" or "readline" request by calling "getchar" for one character at a time. A Boolean value is returned by "getchar" to tell if the character terminates the current line; if so, the character returned is either a newline, control-D, control-W, or null. The first three of these are appropriately interpreted (depending on whether the command is "read" or "readline"); the meaning of a null is explained in Section 2.8. At most MSLEN characters may be read at a time; thus the reply will fit in one message.

2.3 Overview of output

The integer arrays "codes" and "bytesleft", each of size NUMCODES, keep track of output links. A zero entry in "codes" indicates an unused link; otherwise links are given unique codes. When a link is opened, the corresponding entry in "bytesleft" is set to zero; when a write message is received, the indicated length of the write is placed in "bytesleft". Subsequent messages over this link are interpreted as data to be written until the write is completed. The routine "writemsg" is used to write each portion; characters are written with the routine "sayfull" (Section 2.5), except for carriage returns and newlines, which are written directly by calling "sayit".

2.4 Requesting and changing console modes

The variable "modes" stores the current modes. When modes are requested or changed, the routine "showstate" prints out the "current" or "new" modes, respectively. The modes that can be turned on and off are ECHO, HARD, UPPER, and TABS. A HARD terminal cannot backspace its cursor legibly; and UPPER terminal cannot enter lower case directly, and a TABS terminal has hardware tabs.

2.5 Output buffer manipulation

The terminal driver uses a circular output buffer, "ttoutbuf", of size TTOUTBUFSIZE (150). Two variables are used as indices into "ttoutbuf": "nxtoutch", which points to the next available place to put a character into the buffer, and "outbufp", which points to the next character to take out. When these indices are equal, the buffer is empty.

The routine "ttoutdriv" is called at interrupt level to write a character. This routine returns without any action is "paused" is true (Section 2.7). If "ttoutbuf" is nonempty, the next character is written to the terminal, with a delay specified in absolute location 157702 (for debugging and connection to a slow Unix port). After it is displayed, each newline is replaced in the buffer by a carriage return rather than being removed; by this device, carriage returns are effectively appended to newlines. Input interrupts are disabled while messing with the buffer (in case ECHO mode is on).

The routine "sayit" puts one character into "ttoutbuf". If doing so would fill the buffer, "sayit" waits a second and tries again. The variables "outpos" and "tabplptr" are significant for echoing input; "sayit" sets them both to zero after a carriage return or newline. In other cases "outpos" is incremented, except that after a backspace it is decremented. While the buffer is being changed, input and output interrupts are disabled (we might be in ECHO mode).

The routine "sayfull" converts a character into a readable

(or audible) form and calls "sayit". If UPPER mode is on, a "!" is placed before appropriate characters. Except for control-G (bell), control characters are converted to the notation "^A", etc. A rubout is converted to "^#". The array "tabplace" is used to store the cursor position just before each tab, to allow backspacing over tabs; "tabplptr" is an index in "tabplace". At most TABNUM (10) tabs are stored. If TABS mode is off, a tab is converted into several blanks, until "outpos" becomes a multiple of 8 ("sayit" increments "outpos"). If TABS mode is on, the tab character is sent directly to "sayit", and "outpos" is adjusted accordingly.

The routine "sayback" is used when the console is in ECHO but not HARD mode. It converts a given character into several backspaces, to undo the effect of "sayfull", and calls "sayit". Two backspaces are required for control characters, (escaped) rubout, and the UPPER mode escape sequences, except that none are needed for bells. Other characters take one backspace, except for tabs, which require a sequence of backspaces until "outpos" has been decremented (by "sayit") to the appropriate value found in "tabplace". Also, "tabplptr" is decremented.

2.6 Input buffers

The terminal driver uses a circular input buffer, "ttinbuf", of size TTINBUFSIZE (100), and a circular buffer, "lineptr", of size TTLINES (20). The entries in "lineptr" are indices in "ttinbuf" that tell where lines begin; thus TTLINES is the maximum number of lookahead lines. There are two other variables

used as "ttinbuf" indices: "inbufp", which tells the next available spot to put a character into "ttinbuf", and "nxtchar", which tells the next character to take from the buffer. When "nxtchar" is one buffer location ahead of "inbufp", the buffer is full. To permit intra-line editing, lines can only be removed from the buffer when they have been "terminated". There are two variables used as "lineptr" indices: "lastline", which tells the current line being put into the buffer, and "nxtline", which tells the line being removed. The buffer is empty when "lastline" and "nxtline" are equal.

The routine "ttindriv" is called at interrupt level to read a character. If "paused" is true, then only control-Q and control-S will have any effect; all other characters are ignored (Section 2.7). The Boolean variable "escaping" is true when the next character is to have no special meaning; it becomes true when the escape character is read and becomes false after the following character. A character with no special meaning is placed in the buffer by calling "putinbuf"; if the buffer is full, the character is discarded and a bell is written by calling "sayit". If there is room and ECHO mode is on, the character is written by calling "sayfull"; for example, an escaped newline will echo as "^J", which allows backspacing over it later. If UPPER mode is on, appropriate translation takes place.

Various characters cause intra-line editing. A control-C is echoed as the sequence

^C<bell><newline>

and isn't placed in the input buffer; see Section 2.8. An ERASE character (rubout) causes the last character of the present line to be removed from the buffer, using the routine "tkoutbuf". If the line wasn't empty and ECHO mode is on, the character removed from the buffer is echoed, using "sayfull" in HARD mode and "sayback" otherwise. In HARD mode, backslashes ('\') are placed around a sequence of erased characters; when erasing begins, a backslash is echoed and the Boolean variable "erasing" becomes true; when erasing ends, a backslash is echoed and "erasing" becomes false. A KILL character (control-X) removes the entire present line from the buffer. In ECHO mode, "??" is printed. In HARD mode, a newline is also printed. If we are in ECHO but not HARD mode, the KILL is treated as a sequence of ERASEs, until the current line is empty; thus the screen cursor returns to the point where the line began.

A line is "terminated" by an (unescaped) control-D, control-W, carriage return, or newline. In the first two cases, the character is put in the buffer but not echoed. In the last two cases, a newline is put in the buffer and echoed. The routine "termline" updates "lineptr"; if this buffer is full, the line termination is ignored. Whenever a line is terminated, the variable "linecount" is reset to 0. This variable keeps track of how many lines have been output to the terminal since the last time the user entered a line. An "awaken" call is made in case "getchar" (described next) was waiting for the buffer to become nonempty. If this awaken is received in the terminal driver's

main loop, it is properly ignored.

The routine "getchar" takes a character out of "ttinbuf" and also returns a Boolean value to tell if a line has just been completed. (Thus, for example, escaped and unescaped newlines are distinguished.) When a line is completed, "lineptr" is appropriately updated. If the buffer was empty, "getchar" waits for a message on channel 10 to indicate that the interrupt level routine "ttindriv" has put a line into the buffer. (This message might also indicate a control-C; see Section 2.8.) Input interrupts are disabled when the buffer is in an awkward state.

2.7 Pause control

Pause control uses the commands control-S and control-Q. Initially, "scroll" is false. If a control-S is typed in this state, "scroll" is set to true and "pause" is also set to true. The effect is that output pauses until released, and it will continue to periodically pause every SCROLLLEN (18) lines. When the terminal is paused, a control-S will cause it to be released for the next 18 lines, but a control-Q will release it and turn off scroll mode, so it will not stop again. Control-Q can also be used to turn off scroll mode even if the terminal is not currently paused.

2.8 Control-C actions

The variable "killlink" contains the lifeline along which a kill should be performed upon receipt of a control-C. Initially, "killlink" is set to -1 to indicate the absence of such a lifeline. The resource manager encloses such a lifeline in a "TO-KILL" message to the terminal driver, received on channel 1. When a lifeline is received, the Boolean variable "ctrlC" is set to false.

The interrupt level routine "ttindriv" notices when a control-C is typed. If "ctrlC" was false and "killlink" was non-negative, "ctrlC" is set to true and a message is sent to the terminal driver on channel 10 with an "awaken" call. While "ctrlC" is true, all messages received on channels 2 and 3 are ignored and any links enclosed in such messages are destroyed.

The message on channel 10 is received either in the terminal driver's main loop or in the routine "getchar", which was waiting for a non-empty input buffer. In either case, the routine "chkctrlC" performs the kill if "ctrlC" is true, flushes all outstanding messages on channel 10, and reinitializes "killlink" and the input and output buffers. Any lame-duck messages on channels 2 and 3 will be ignored because "ctrlC" is still true. A Boolean value is returned by "chkctrlC" so that "getchar" knows that the message was a control-C indicator rather than a non-empty buffer indicator.

After a kill is performed, control must return to the main loop. If the message had been received inside "getchar", the

value returned is the null character as a line terminator. This special case is recognized by "readmsg" which then returns to the main loop without completing its read request; the link enclosed with that request is destroyed.

2.9 Pausing and continuing

When an unescaped control-S is read, the variable "paused" is set to true. When an unescaped control-Q is read, that variable is reset to false and the output interrupt enabling is toggled to restart the output-interrupt driven routine "ttoutdriv". That routine returns without any action if "paused" is true.

Files

ttdriver.u, ttdriver.h, filesys.h

Data Structures

char ttinbuf[TTINBUFSIZE]
: Circular input buffer filled by ttindriv, emptied by readmsg.

int lineptr[TTLINES]
: Circular buffer of ttinbuf indices that point to beginnings of lines.

char escaping
: Boolean; set by ESCAPE, reset by next character.

char erasing
: Boolean; true during a sequence of ERASEs; only used in hard copy mode.

char modes
: Bits used: ECHO, TABS, HARD, UPPER.

int tabplace[TABNUM],tabplptr
: Remembers where tabs were.

char ttoutbuf[TTOUTBUFSIZE]
: Circular output buffer. Filled by sayit, emptied by ttout-driv.

int codes[NUMCODES]
: Currently active output lines.

int bytesleft[NUMCODES]
: Used to keep track of pieces of different write messages.

int killlink
: Tells the ttdriver whom to kill on ^C.

Procedures

main(dev)
: Initializes tables, provides parent with a request link,

prepares to use terminal whose device register is at "dev". Executes a loop that receives and dispatches client requests.

readmsg(len,how)
Reads "len" characters, using routine "getchar". At most MSLEN characters are read. Reading terminates if a control-D is read. In the case that "how" is READLINE, any line terminator (control-W or <cr> or <lf>) terminates reading.

char getchar(ch) char *ch;
Gets a character from the input buffer and returns it in "ch". The returned value is Boolean: TRUE means the character returned ends a line.

ttindriv()
Called by "ttinint" when a character is ready; runs at interrupt level. Reading a control-C causes an "awaken" service call, ERASE or KILL cause intra-line editing. Line termination is caused by control-D, control-W, <cr> (converted into <lf>) and <lf>. Termination causes an "awaken" service call. The input buffer is updated, and the input is properly echoed.

char putinbuf(ch) char ch;
Puts the given character into the input buffer. It returns TRUE only if there was room in the buffer.

char tkoutbuf(ch) char *ch;
Removes last character of current line from buffer. Returns TRUE only if something was there. The character is returned in "ch".

termline()
Called at interrupt level to cause an "awaken" and to reset buffer pointers.

resetbuf()
Removes the current line by resetting an input buffer pointer.

writemsg()
Decodes a write message from a client. If it is the header of several packets with data, variables are initialized to receive the data. If data have arrived, they are placed in the output buffer by "sayit" and "sayfull".

char chkctrlC()
If a control-C has been received, all awaken messages are flushed, a service call "kill" is performed along the killlink, and the routine returns "TRUE".

closeinput()
Reduces the count of input lines in use.

closeoutput()
Resets the appropriate output line information.

openline(how)
Handles a client request for a new input or output line, as described by "how". Appropriate variables are initialized.

reply(retcode,size) char retcode;
Reports "retcode" to the current client. The "size" parameter tells how much of the standard message buffer has been filled with other useful information that the client must

also receive. The reply code is put in the first byte.

showstate(when) char *when;
Prints the current modes on the terminal with an introductory message determined by "when".

sayit(ch) char ch;
Puts one character in the output buffer and adjusts position variable "outpos" accoringly. This routine is used both for input and output echoing.

sayfull(ch) char ch;
Uses "sayit" to provide a readable form for any character according to the current modes.

sayback(ch) char ch;
Prints as many backspaces as necessary to obliterate the full printing of character "ch" under current modes.

ttoutdriv()
Called at interrupt level. Waits a standard delay to slow down the terminal and then sends one character from the output buffer to the terminal. Line feeds are followed by carriage returns. Returns with no action if "paused" is true.

ttyflush()
Removes any character waiting in the terminal input buffer.

inflush()
Clears out the entire input buffer.

outflush()
Clears out the entire output buffer.

3. THE COMMAND INTERPRETER

3.1 General

The command interpreter is a FOREGROUND process that executes commands typed at its console. The command interpreter may start another FOREGROUND process, which communicates with the command interpreter to get command-line arguments.

The command interpreter is compiled by executing "makecomint", which compiles and links together three files to produce "comint". The three source files are: "comint.u", which handles command line parsing, "comutil.u", which contains routines to execute most commands, and "comrun.u", which executes the "run" command.

3.2 Initialization

The command interpreter acquires a file manager link, a terminal driver link, and terminal input and output links from the resource manager. The terminal driver link is only used to request or change console modes; initially, the command interpreter sets these to "ECHO".

3.3 Command line parsing

A line is input with a "readline" call and converted into a null-terminated string. The line is truncated to LINEMAX-1 characters (LINEMAX is 200).

The routine "findargs" scans the input line, separating it into arguments. Sequences of characters enclosed in quotes are left alone, with the quotes deleted. The Boolean variable "quoted" is true during this process. Two consecutive quotes encountered while "quoted" is true are converted into one quote and do not turn off "quoted". When "quoted" is false, a blank or tab is converted into a null to terminate an argument. Any immediately following blanks or tabs are ignored; the Boolean variable "spacing" is true during this process. At the beginning of the line, "quoted" is false and "spacing" is true. The array "argvec" returns pointers to the argument locations; an entry is made in "argvec" when "spacing" changes from true to false. The variable "argcount" tells the number of arguments; it is incremented when "spacing" changes from false to true or at the end of the line if "spacing" is false.

If there are no arguments, no action is taken. Only the first MAXARGS (10) arguments are used; the rest are ignored.

The routine "lookup" searches a list of character strings to find those whose initial segments match a given string argument. The list format is an alphabetically sorted array of character strings alternating with corresponding codes (integers), and with pseudodata sentinels at each end. Two pointers into the table, "low" and "high", start at opposite ends and move toward each other as the argument is scanned. As each character in the argument is examined, "low" moves up the table so long as this character is larger than the corresponding ones in the table at which "low" points; "high" does the reverse. The process stops if the argument is exhausted or if "high" and "low" pass each other. In the latter case, there is no match. In the former, there are one or more matches; "low" and "high" are equal or not accordingly.

The first argument on the command line is deciphered as a command by calling "lookup" with the table "commands". If there is a unique match, the appropriate action is taken.

3.4 Command execution

The "background" command starts a process with modes "BACKGROUND" and "REUSE" and passes the given argument as an integer. An answer is received from the resource manager and the new process's process identifier is printed. The new process is not given a link to the command interpreter.

The "copy" and "type" commands are translated into "run copyfile" commands. The program "copyfile" is an independent program

that copies one file to another, with the terminal as the default for the second file.

The "directory" command executes a "stat" on the indicated file and prints selected portions of the information returned.

The "make" command "creates" the indicated file and performs "read" commands for IOBUFSIZE (100) bytes at a time from the terminal. After each "read", a "write" is done to the file; finally, the file is closed. The end is indicated by a "read" returning less than 512 bytes; thus, if the input has exactly 512 bytes (or a multiple thereof), it must be terminated by an extra control-D.

The "run" command starts a process with modes "FOREGROUND" and "REUSE" and passes as an argument the number of command line arguments. The resource manager is given a REQUEST link for the child and the terminal input link is closed so that the child may open it. Command line arguments are sent to the child when requested. The command interpreter assumes that the child has terminated when the REQUEST link is destroyed; it then reads the next console command. If the start fails, the command interpreter waits for the REQUEST link to be destroyed before continuing.

The "set" or "SET" command first requests the current modes, which causes the terminal driver to print them. The command line arguments are then deciphered individually; a "-" prefix is remembered with the Boolean variable "notflag" and the command itself is decoded by calling "lookup" (Section 3.3) with the table "modetab". When a mode is recognized, the current mode

specification is altered accordingly. Finally, the modes are changed, and the terminal driver prints the new modes.

The "time" command, if given an argument, sets the time by calling "datetol" and "setdate". The argument is only checked to see that it has ten characters, and zeroes are added for the number of seconds. With or without an argument, "time" finally prints the current time, which is done by calling "date" and "ltodate".

Files

comint.u, comutil.u, comrun.u, comint.h

Data Structures

char *commands[]

Holds the known commands paired with an internal distinguishing code. The array must be in alphabetical order.

char *argvec[MAXARGS]

The arguments to started processes are stored here.

char *modetab[]

A table of terminal mode names to be used with the routine "lookup".

Procedures

main()

Initializes tables, acquires file manager and terminal driver links, then executes a loop that accepts commands from the terminal and dispatches them.

int findargs(line) char *line;

"Line" is null-terminated (without final newline) and doesn't contain any embedded nulls. Puts pointers to the beginnings of arguments into the array "argvec" and the count of how many were found into the global "argcount". Terminates the arguments with nulls. Spaces and tabs are considered delimeters unless they appear in quotes ("). Two consecutive quotes inside quotes are considered one quote. Other quotes are stripped.

lookup(str,table,tablesize,result) char *str, **table; int *result;

Looks up character string "str" in "table". Sets result[0] and result[1] such that table[result[0]], table[result[0]+1], ... , table[result[1]] all have "str" as an initial segment. If result[0]>result[1], there was no match. Assumes that table[0], table[2], ... are strings kept sorted in alphabetical order, and table[1], table[3], ... are other data to be ignored in lookup. Assumes further that table[0] is guaranteed to compare low and table[tablesize-2] is guaranteed to compare high with "str".

int intype(fname) char *fname;
Handles a "make" command. Accepts input from the terminal, creates a new file with name "fname" and puts all input on that file. Returns 0 on success, -1 on failure.

dir(fname) char *fname;
Handles a "dir" command. Uses the file manager to read the directory information from a file, and prints it on the terminal.

setmodes()
Handles a "set" command. Uses "lookup" to find what modes are requested, and communicates with the terminal driver to establish those modes.

printtime()
Handles a "time" command. Finds the current time with the service call "date" and the library routine "ltodate", then prints the result.

settime(s) char *s;
Handles a "time" command with an argument. Uses the library routine "datetol" and the service call "setdate" to change the kernel's date.

int runback(fname,arg) char *fname;
Attempts to run the file "fname" as a background process, handling the "back" command. It returns the process id of the new process or -1 on failure.

killback(procid)
Sends a note to the resource manager to kill the process whose identifier is "procid". Handles the "kill" command.

int run(fname,argc,arg0) char *fname;
Handles the "run" command. Attempts to load and run the executable file named by "file". Returns 0 on success. If "argc" > 0 then uses "arg0" and following arguments to satisfy requests for arguments instead of arguments from the command line. Executes a loop that waits for requests from the child for arguments until the child terminates.

4. THE FILE MANAGER

4.1 General

The file manager forwards requests from other processes to the demon running on the PDP-11/40 where they are implemented under Unix. (See Section 5 for details on the demon.)

The code lies in "filesys.u"; all processes using the file manager should include "filesys.h". (It isn't necessary to in-

clude "filesys.h" if all communication is done by library routines.)

The word-parallel line used to communicate with the PDP-11/40 uses three registers at location DR11.40 (octal 167770). All reading and writing use busy waits. More details are found in the file "io.h".

The file manager initially gives a REQUEST link to its parent (usually the resource manager) with channel 1. All "open", "create", "alias", "unlink", and "stat" requests come over this link or copies thereof. When a file is "opened" or "created", a new link with channel 2 is enclosed with the reply. This link will be used for "read", "readline", "write", and "seek" requests; its destruction indicates a "close" request.

4.2 Execution of requests

The file manager executes most requests by receiving a message from the client, writing a request over the word-parallel line to the PDP-11/40, reading the reply from the word-parallel line, and sending it to the client.

Requests on channel 1 contain file names. These are communicated over the word-parallel line by first writing the length and then the name. The routine "rawopcr" is used by "open", "create", "alias", and "unlink" to send a request to the demon and receive a one-word reply to be forwarded to the client. In the cases of successful "open" or "create" calls, a new link with channel 2 is enclosed with the reply; the code for this link is the same as the value returned to the client (a Unix file

descriptor). This new link is of type GIVEALL but not DUPALL. The routine "rawstat" is used by "stat"; it reads 38 bytes from the demon. The first word tells whether the stat was successful; either 0 or 36 bytes are forwarded to the client accordingly.

A request on channel 2 refers to an open file; the file descriptor for this file is the code of the link. The routine "rawread" forwards a "read" or "readline" request to the demon and reads the reply. An integer telling how many bytes were actually read comes first, followed by the bytes themselves. The bytes read are then forwarded to the client. No more than MSLEN bytes should be read at a time, so that one message suffices for the reply. The routine "rawseek" similarly treats "seek" requests, except that only one word is read from the demon, and then forwarded to the client.

When a file is opened for writing, the corresponding entry in the array "bytesleft" is set to zero. ("Bytesleft" is indexed by file descriptors.) When a "write" request is received on channel 2, the indicated length for the write is inserted in "bytesleft". Further messages on the same link are taken as data to be written (MSLEN bytes at a time) until the write is completed. As each portion is received, the routine "rawwrite" sends it to the demon and waits for an acknowledgment before proceeding. No reply is given to the client.

When a link on channel 2 is destroyed, a "close" message is sent to the demon. No response is read from the demon, and no reply is made to the client.

Files

filesys.h, filesys.u, io.h

Procedures

main()

Initializes tables, then executes a loop awaiting client requests and dispatching them.

rawstat(name,replylink) char *name;

Handles a "stat" request. The file "name" is sent to the demon. Its answer is returned to the client; failure is marked by an empty message.

int rawopcr(file,mode,how) char *file;

The argument "how" is OPEN, ALIAS, UNLINK, or CREAT. A message is sent to the demon to do the appropriate action to "file". The "mode" is the same as Unix mode for files. The file descriptor given by the demon is returned.

rawread(rwfd,buf,bytes) char *buf;

Reads "bytes" bytes from the file whose descriptor is "rwfd" into "buf", which must be on a word boundary (even).

readdr(buf,rwlen) int *buf;

Reads ceiling(rwlen/2) words from the DR line to Unix into "buf", which must be word-aligned (even).

writedr(buf,rwlen) int *buf;

Writes ceiling(rwlen/2) words to the DR line to Unix from "buf", which must be word-aligned (even).

rawclose(usrcode)

Handles a client "close" request. Sends a note to the demon to close the file whose descriptor is "usercode".

rawwrite(rwfd,buf,bytes) char *buf;

Handles a "write" request from a client. Gives the demon data from "buf" of length "bytes" to be placed in file "rwfd".

int rawseek(skfd,offset,mode)

Handles a "seek" request from a client. Sends a note to the demon to do the given seek ("offset" and "mode" mean the same as in Unix) to file identified as "skfd". Success returns 0; failure -1.

5. THE DEMON

5.1 General

The "demon" is a program that runs on the PDP-11/40 under Unix. Its code is in "demon.c". Roscoe processes that communicate with it must include "demon.h".

For each LSI there is an associated demon. This demon reads from a word-parallel line connected to that LSI; the Unix names

for these lines are "/dev/drx", where x = 0,...,4. Commands are translated into corresponding Unix system calls and appropriate responses are written to the word-parallel line. All user process communication at the LSI side of the fence is done by the file manager (Section 4).

Each message sent in either direction on the word-parallel line is preceded by at least one header word of "NONSENSE" (octal 125252). After the header word(s), the next three words of a message to the demon have the following structure:

```
struct {
     int command,code,length;
}
```

The number of bytes remaining in the message is "length". These remaining bytes are usually a file name, in which case they will subsequently be read into the character array "fname", of size MAXNAME (40). The value returned over the word-parallel line is usually a single word (after a word of "NONSENSE").

The demon sits in an infinite loop awaiting messages. When a message is received, the appropriate action is taken, as described in further subsections. The routine "getstr" is used to read from the word-parallel line; it rounds the number of bytes up to an even integer and watches out for errors due to terminal interrupts. The routine "signal" is called to catch terminal interrupts, which otherwise plague all Unix processes started at a given terminal.

5.2 DALIAS command

The string "fname" is split into two pieces to become the two arguments for the Unix call "link". The length of the first substring is "code". The effect of "link" is to make its second argument an alias for the first one. The value returned by "link" is passed on.

5.3 DCLOSE command

file descriptor "code" is closed (Unix call "close"). No message is returned.

5.4 DCREAT command

The file "fname" is created (Unix call "creat") with mode "code". The value returned by "creat" is passed on.

5.5 DOPEN command

The file "fname" is opened (Unix call "open") with mode "code". The value returned by "open" is passed on.

5.6 DREAD command

For this command, "length" tells the number of bytes to read from file descriptor "code", using the Unix call "read". This length is truncated to "BUFLEN" (512). The first word of the return message is "code". The second word is the value returned by "read", which tells the number of bytes actually read. The bytes read are written next; if "read" returns -1 (error), nothing else

is written. A garbage byte will exist at the end if the number of bytes actually read was odd.

5.7 DREADLINE command

This command is identical to "DREAD", except that the Unix call "read" is used for one byte at a time. If a "newline" character is encountered, it is considered part of the returned text, and reading stops.

5.8 DSEEK command

A Unix call "seek" is performed on file descriptor "code". The offset for the "seek" call is "length"; the mode for the "seek" call is the next word read from the word-parallel line. The value returned by "seek" is passed on.

5.9 DSTAT command

A Unix call "stat" is performed on file "fname". The first word of the return message is -1 for failure, 36 for success. In either case, 36 additional bytes are written; if the "stat" succeeded, these bytes are the desired information.

5.10 DTIME command

A Unix call "time" is performed to return a double word. The first word of the return message is 0; the next two words are the result of the "time" call. This command is only used by the resource manager during Roscoe initialization.

5.11 DUNLINK command

A Unix call "unlink" is performed on the file "fname". The value returned by "unlink" is passed on.

5.12 DWRITE command

The rest of the incoming message is read into "writebuf"; the length of this text is "length", truncated to BUFLEN (512) bytes. This text is then written to file descriptor "code", using the Unix call "write". The value returned is "code" if "write" returned success; otherwise, the value returned is "code" times minus one.

6. LIBRARY ROUTINES

All the files in this section are in the directory "/usr/network/roscoe/library". The object code is archived in "libr.a".

6.1 File manager routines

These routines communicate with the file manager (Section 4).

The routine "opcreat" is used by "open", "create", "alias", "unlink", and "stat" (the sources reside in "opcr.u", "open.u", "create.u", "alias.u", "unlnk.u", and "stat.u", respectively) to send a command and file name to the file manager over the given file manager link. Another argument, "mode", has various meanings for "open", "create", and "alias" calls. In the case of

"stat", an additional argument represents a REPLY link that is passed to the file manager. The routine "stat" receives a response over this link and copies the information into the designated buffer. In the other four cases, "opcreat" waits for a response from the file manager and gives a return value accordingly (this value is a link number after a successful "open" or "create"). The routine "alias" concatenates its two file name arguments before calling "opcreat"; "mode" is then the length of the first name, to eventually be decoded by the demon.

The routine "close" (in "close.u") is synonymous with "destroy".

The routine "someread" (in "somerd.u") is used by both "read" and "readline" (in "read.u" and "rdln.u", respectively). The appropriate command is sent over the given link (to either the file manager or terminal driver). All reads are split up into individual requests for MSLEN bytes at a time. The responses for the portions of the read are copied into the given buffer.

The routine "seek" (in "seek.u") sends an appropriate message to the file manager over the given link, and waits for a reply. Success is reported by a zero in the first word of the reply message.

The routine "write" (in "write.u") sends the appropriate header message over the given link (to the file manager or terminal driver) and then sends the data in subsequent messages MSLEN bytes at a time. No reply is awaited, and no value is returned. The routine "print" (in "print.u") is similar to the Unix printf routine. It edits the output string and calls "write". A linear

buffer, "prbuf", of size PRINTBUFSIZE (100), is used. The format string is scanned and the routines "printint", "printlong", "printoct", and "printstr" are called to handle the conversions for "%d", "%w", "%o", and "%s" format items, respectively. Both "printint" and "printlong" check for the sign, take absolute value, and use division by 10 (recursively), although "printlong" uses long arithmetic. The routine "printoct" always produces six characters; it first checks the sign bit, and then inspects three bits at a time with an appropriate shift. In all cases "printch" is used to put characters into "prbuf" and to call "write" when the buffer is full. The buffer is also flushed at the end.

6.2 Resource manager request routines

The routines "fsline" and "parline" (in "fsline.u" and "parlin.u") ask the resource manager (Section 1) for the appropriate link and return the enclosure. The routines "inline" and "outline" (in "inline.u" and "outlin.u") first ask the resource manager for a terminal link, then ask the terminal driver for the appropriate line, and finally return the enclosure.

The routine "fork" (in "fork.u") sends a start message to the resource manager, conveying the file name, argument, and mode, but always specifying ANSWER. A link is given to the child of type REQUEST, GIVEALL, and TELLDEST. The first word of the reply message is returned; in particular, this word is the process identifier for a BACKGROUND child. If the start failed, "fork" waits for the given link to be destroyed.

The routine "killoff" (in "killof.u") conveys the kill request to the resource manager, gets a reply, and returns the first word of the reply message.

6.3 Roscoe service calls

The Roscoe service call interface is the assembler file "lib.s". For each call, an appropriate magic number is placed in register 1 and a jump is made to the Roscoe entry point "sys" (octal location 1002). Arguments are left on the stack; the kernel takes it from there.

6.4 Miscellaneous

The routine "atoi" (in "atoi.u") converts a string into an integer.

The file "call.u" contains "call" and "recall". The routine "call" sends a message as indicated, encloses a REPLY link, puts the REPLY link's code into the global variable "unique", and invokes "recall". The latter receives a message with a five second delay, and checks that the incoming message has the proper code ("unique") and note ("DATA").

The assembler file "reset.s" contains "setexit" and "reset". The routine "setexit" saves register 5 and the old program counter in global locations "sr5" and "spc". The routine "reset", by restoring these, effects a return to the environment which last called "setexit".

The file "user.h" contains various macros freely referred to in this documentation. For the user's convenience, it also de-

fines TRUE (all 1's) and FALSE (all 0's) and the following structures:

```
struct {char lowbyte,highbyte;};
struct {int word1,word2;};
```

The file "time.u" contains the routines "datetol" and "ltodate", which convert character strings into long integers (representing seconds since the beginning of 1973) and vice versa, respectively. The array "calendar" contains the number of days preceding each month in a leap year, with pseudodata "366" as a thirteenth entry. Character string arrays store the days of the week and months of the year. The macro FOURYEARS gives the number of days in a four-year period. The first step of "datetol" is to convert the given string, with format "yymmddhhmmss" into an array of six two-digit integers. Arrays "lbound" and "ubound" are used to check that these integers are reasonable. Sizes of months are also checked by subtracting the appropriate consecutive entries in "calendar". The number of days is calculated by computing the number of four-year intervals beginning with 1973 (and multiplying by FOURYEARS), then adding on the proper (0-3) number of (non-leap) years (times 365), then adding on the month offset as found in "calendar", and finally adding in the day of the month. For non-leap years, February 29th is caught as a mistake, and any day occurring later in the year is decreased by one. Finally, hours, minutes, and seconds are added on. The reverse process is carried out by "ltodate". Seconds, minutes, and hours are first removed. The day of the week is computed from the number of days modulo 7. Division by FOURYEARS determines the four-year period; the remainder determines the ex-

act year and day within the year, with a remainder of (4*365) repesenting December 31st of a leap year. In a non-leap year, conversion (to the proper format for "calendar") is performed by increasing by one any day of the year larger than 58. (February 28th remains 58; March 1st is bumped to 60; etc.) The month is calculated by dividing the number of days by 30; the answer may be too large by one and is corrected by inspecting "calendar". As the result is computed, it is edited into a character string.

REFERENCES

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

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., Solomon, M. H., Finkel, R. A., ROSCOE User Guide, University of Wisconsin -- Madison Computer Sciences Technical Report #336, September 1978.