KAN

Unirex

System Description

Preliminary

Dansk Data Elektronik ApS

27 July 1982

Author: Claus Tøndering

1. Introduction.

This manual contains a preliminary description of the Unirex operating system running on the Unimax computer. It describes the various concepts and system directives available, but does not include the utility program which will be available.

Numbers starting with a ´$´ character are hexadecimal, other numbers are decimal.

Unirex and Unimax are - hopefully - registered trade marks of Dansk Data Elektronik ApS. UNIX is a trade mark of Bell Laboratories.

3. Bootstrapping.

One disk in the computer is termed the ´boot disk´. This is the disk, from which the operating system will be loaded when power is applied to the computer.

The boot disk must have a Unifile file system structure.

On the boot disk a number of files will be present. Some of these files contain self-test programs for the various CPUs in the computer, other files contain the actual operating system.

When power is applied to the Unimax, a number of self test programs are automatically run. After this the following files are loaded:

```
/unirex   is loaded into the master CPUs.
/unifile  is loaded into the master CPUs.
/sioc     is loaded into the SIOCs.
/dioc     is loaded into the DIOCs.
```

The master CPUs inspect the file /configur, which tells the operating system, which channels on the SIOCs are terminals, and which are printers. (Additional information may at a later time be stored in /configur.)

The master CPUs start /unifile program on behalf of user number 1.

Master CPU number N now executes the following subroutine call (see section 6.6.1) on behalf of user number 0:

```
char *parms(.2.);
parms (.0.) = "/initN"
parms (.1.) = "N"
ld_prod("/initN",0,10,2,parms,0,0,0,&errblock);
```

The N in the above lines is the CPU number.

Typical duties of /initN may be:

- set the terminal and printer characteristics (only /init1).
- load and execute Mikfile.
- mount various disks.
- change access rights for various devices (only /init1).
- install various often-used programs (especially the logon program).

- initiate a spooler process.
- display start-up message on all terminals (only /init1).
- put all terminals in log-on mode (only /init1).

Alternatively, /initN may just interpret a command file ´/startN´ specifying what should be done.

4. Gaining Access to the System.

When the Unimax has been bootstrapped all the terminals will probably display a log-on message such as, for example:

Unirex is ready for log on. Please press the escape key:

This is an invitation to the user to log on to the computer, that is, to acquire access to a master CPU. But in order for this to be possible the user must be authorized to do so.

4.1. User Number, Name, and Password.

Each user authorized to use the computer is assigned a user number in the range $0000 to $FFFF and a name of up to 8 characters. The user may assign himself a password of up to 8 characters.

The name and the user number are synonymous; the user number is used to identify the user within the computer, the name is the means by which the user identifies himself to the computer. Using the name here instead of the user number has the advantage that a name is usually easier to remember than a number, and also a name normaly contains redundant characters decreasing the possibility of erroneous input.

The password is a secret code word which the user assigns himself to prevent misuse of his access right to the system.

The user number determines which devices and files in the system the user may access. Further, the user number is used for accounting purposes: The system stores information about how many times and for how long the user has used the computer.

The first 2 hexadecimal digits of the user number specify the ´group´ to which the user belongs. This is used in the I/O unit protection scheme.

4.2. Privileged Users.

Users in group 0 are ´privileged´. Privileged users have certain rights in the system, which unprivileged users do not. For example, privileged users may access any device or file in the computer, they may authorize new users to use the computer, and they may abort programs running in the computer, regardless of who started the program.

4.3. The Access File.

A file called the ´Access File´ is present on some disk in the computer. In this file information about all the users that have access to the system is stored. The information in this file is:

- User number.
- User name.
- User password (encrypted?).
- Number of times the user has logged on to the computer.
- Total logged on time.
- Date and time of last use of the computer.
- Name of a program the user wishes to be executed immediately after log on.

This file is protected so that it may only be accessed by privileged users.

The file is a text file, and may thus be inspected and changed by an editor program.

4.4. Logging On.

The following text is presented on a terminal, when nobody uses it:

Unirex is ready for log on. Please press the escape key:

This is an invitation to the user to log on to the computer, that is, to request access to a master CPU.

When the user presses the escape key, the text ´<ESC>´ will appear on the screen, and the terminal driver program (which operates the terminal controller) examines the contents of common memory to see which master CPU has the fewest users logged on. The terminal driver then tells that CPU that a log on is requested, whereupon the master CPU goes through the log on procedure as described below.

The user may, alternatively, request to be logged on to a particular master CPU, for example, if he wants to abort a program he knows to be executing on that particular CPU. The user enters the CPU number and presses the escape key, whereupon the terminal driver will request a log on on the master CPU having the number given by the user.

The operating system in the master CPU now performs the following call on behalf of user 0:

```
char *parm(.2.);
parm(.0.)="/logon";
parm(.1.)="/termNN";  /* this is the device from which a logon
                         is requested */
if ("logon" is installed) then
  in_prod("logon","logonNN",10,2,parm,0,0,0);
otherwise
  ld_prod("/logon","logonNN",10,2,parm,0,0,0,&errblock);
endif;
```

The logon program typically performs duties such as:

- acquire user name and password.
- check that the user has access to the system.
- spawn whatever program that has been specified in the access file.
  (At this point the user program executes.)
- wait for the completion of the offspring process.
- write accounting information into the access file.å
- dispaly logon request message on terminal.
- put terminal in logon mode.

5. The Supervisor.

The Supervisor is the main part of the Unirex operating system in the master CPUs. The supervisor controlls the execution of the user programs and supplies the user with various services.

The functions of the supervisor may be divided into three main catagories:

1) Process management. (Described in chapter 6.)
2) Memory management. (Described in chapter 7.)
3) I/O management. (Described in chapter 8.)

5.1. System Calls.

The following chapters describe a set of system subroutines in C. These system calls issue the various supervisor requests. They all return an error code, which is zero if no error occured.

These subroutines will be supplemented by a set of auxiliary subroutines that facilitate the use of the system calls, plus a set of subroutines modelling UNIX system calls.

6. Process Management.

A process is a running program. Processes have various properties, and various operations may be performed on processes.

A process is said to ´belong´ to the user who started it, and, conversely, he is said to ´own´ the process.

A process han an ´effective user number´ and a ´real user number´. The effective user number determines the access rights and privilege rights of the process. The real user number determines who is allowed to perform operations on a process. Operations on a process can only be performed by the user whose user number is identical to the real user number of the process or by a privileged user.

The real user number is always the user number of the user starting the process. The effective user number is normally identical to the real user number; if, however, the file containing the program has the ´set user id´ bit on, the effective user number will be the number of the user owning the program file.

A process may be identified by either of the following:

1) Its name, which is a string of 8 characters, and the number of the user owning the process.

2) Its process number.

Thus diffent users may have processes with identical names running simultaneously.

When a process is started, the user may either himself supply a process name, or request the supervisor to do so.

6.1. The Different Kinds of Processes.

A process is either a ´main process´, which is directly subordinate to the operating system, or it may be a ´sub-process´, which is subordinate to another process. The diffence between these two kinds of processes is mainly reflected in what happens when a process dies, that is, terminates its execution. This is described in section 6.5.

The programs for the processes may be located in three different places:

1) In a file from which it is loaded when execution starts. This is termed a ´loaded´ program.
2) In the memory belonging to another process. This is termed an ´in-memory´ program.
3) ´Installed´ in memory, that is, permanently present in the master CPU memory. This is termed an ´installed´ program.

A few words about installed programs are in order: Execution of installed programs can start very fast, because there is no need to load the program. One or more read/write segments for data are simply allocated and the code is executed. In this way the code for an installed program may be used in the execution of several processes. This facility may be used, for example, for the Pascal compiler, the Pascal interpreter, the Comal interpreter, the assembler, the various utility programs, etc. Only privileged users may install programs.

6.2. Priorities.

Processes have priorities. The priority is a number in the range -20..+20 with -20 indicating the highest priority. Negative priorities may only be used by privileged processes, whereas priorities 0..+20 are available to all users. Most user processes should have a priority of 10.

The priority is used when several processes are competing for access to the CPU.

6.3. Operations on Processes.

6.3.1. Starting a Process.

Program execution may be started in four different ways:

1) By ´Production´.
2) By ´Spawning´.
3) By ´Metamorphosis´.
4) By ´Forking´.

6.3.1.1. Production.

A process, A, may ´produce´ another process, B. In this case B will be a main-process. The program code for B may either be loaded or installed, but it may not be an in-memory program.

A is termed the ´producing´ process. B is termed the ´produced´ process.

6.3.1.2. Spawning.

A process, A, may ´spawn´ another process, B. In this case B will be a sub-process, subordinate to A. The program code for B may be loaded, installed, or an in-memory program.

A is termed the ´parent´ process. B is termed the ´offspring´ process.

6.3.1.3. Metamorphosis.

A process, A, may ´metamorphose´. This means that the program code for A is replaced by another program code, and the execution of A continues with the new program code. All open files will remain open and no completion code is reported, because A is not considered dead, but merely transformed, metamorphosed, into another shape.

The new program code may be loaded or installed, but it may not be an in-memory program. Further, the process requesting metamorphosis may not be an in-memory program nor may it be the parent process of any executing in-memory programs.

6.3.1.4. Forking.

A process, A, may ´fork´. This means that an identical copy of the program code for A is made, and execution continues both in A and in the copy, being process B. B will be a sub-process subordinate to A.

A is termed the ´forking´ process. B is termed the ´forked´ process.

The process requesting forking may not be an in-memory program. If it is the parent process of an executing in-memory program, this in-memory program is not forked together with the parent process.

6.3.1.5. Open I/O Units.

The initial execution environment of a process is largely determined by the I/O units with which it communicates. When a program starts executing it inherits a number of open I/O units from the process that started it. The following possibilities exist:

1) A process is produced or spawned. In this case the producing/parent process specifies which of its own open I/O units that should be passed to the produced/offspring process.

2) A process metamorphoses. In this case the open I/O units remain open when execution of the new program starts.

3) A process forks. In this case all the open I/O units of the parent process are passed to the offspring process.

It should be noted, that this means that two processes may be working on the same file simultaneously. Seeking and inputting/outputting to that file may not be well-defined if the two processed do it simultaneously.

6.3.2. Exiting a Process.

A process may terminate its execution by issuing an exit request. This involves informing its parent process, if any, that it has done so (see section 6.5), releasing all memory belonging to that process, and closing all files which the process has not closed itself.

6.3.3. Aborting a Process.

A user may abort a process, that is, force the process to exit. An unprivileged user may only abort his own processes. A privileged user may abort any process.

6.3.4. Suspending a Proces.

The execution of a process may be temporarily suspended. An unprivileged user may only suspend his own processes. A privileged user may suspend any process.

The execution is resumed when the user issues a ´resume´ request or when a specified time expires.

6.3.5. Installing a Program.

A privileged user may install programs. Such programs must consist only of read-only segments and unitialized read/write segments.

When a program is installed the contents of the read-only segments are read into memory. This memory will be shared by all processes executing this program.

When an installed program is executed, the program code is located in the read-only segments already in memory. Read/write segments will be assigned to the process as required and will not be shared by the different processes executing this program.

An installed program can only be removed from memory by privileged users and only if no process is currently executing it.

6.3.6. The Process Stack.

When a loaded or installed program starts executing, it is given an initial stack size, determined at link time.

When an in-memory program is spawned, the parent process specifies the stack size required by the offspring process. This is taken from the low-address end of the stack of the parent process.

6.3.7. Process Entry.

The main procedure of a program must have the name main, declared in the following manner:

```
main(ac,av)
  short int ac;
  char **av;
```

When the program starts execution, the supervisor will place a possible parameter string on its stack. The program will in ac find the number of parameters. av(.0.) will be the address of a null-terminated string containing the first parameter, av(.1.) will be the address of a null-terminated string containing the second parameter etc. av(.0.) will typically be used to hold the name by which the program was invoked, making av(.1.) the first effective parameter.

6.4. Operator Communication.

No particular operator communication program exists resident in Unirex. When a user logs on to a terminal, the logon process starts whatever program it may find in the logon file. This program may be some kind of operator communication, such as a Unix shell, or, perhaps, something better.

6.5. Process Death.

A proces may die either by committing suicide using an exit request or by being killed by another process issuing an abort request. When a process dies, information about its death is reported to its parent process, if any.

If a parent process dies, all of its offspring processes are automatically aborted.

The death of a main process is not reported anywhere.

When a process dies, it is automatically detached from all partitions, and all its open I/O units are closed.

6.6. System Calls.

The following system calls are used in process management:

6.6.1. Produce Process from Loaded Program.

```
ld_prod(uname,prname,prio,pc,pv,uc,uv,pid,errblock)
  char *uname, *prname;
  int  prio, pc;
  char *pv(..);
  int uc;
  short int uv (. .), *pid;
  char *errblock;
```

This subroutine produces a process from a loaded program.

Parameters:

uname is the address of a null-terminated string being the unitname of the file containing the program.

prname is the address of an 8 character string containing the name which is to be assigned to the process. If prname==0, Unirex assignes the name ´$$$$nnnn´ to the process, where nnnn is the hexadecimal value of the process number.

prio is the priority of the process. It must lie in the range -20..+20 for privileged processes, and 0..+20 for uniprivileged processes.

pc is the number of parameter strings passed to the process.

pv is the address of an array of null-terminated parameter strings to be passed to the process.

uc is the number of open I/O units which the produced process should inherit from the producing process.

uv is the address of an array of uc I/O unit descritors, that should be inherited by the produced process. uv(.0.) is the I/O unit descriptor in the producing process which will become I/O unit descriptor 0 in the produced process. uv(.1.) is the I/O unit descriptor in the producing process which will become I/O unit descriptor 1 in the produced process, etc.

pid is the address of a short integer in which the process number of the produced process will be stored. If pid==0 the process number will not be stored.

errblock is the address of a 6-byte array in which additional error information may be stored by the disk driver if a hard disk error occurs during program load.

6.6.2. Spawn Process from Loaded Program.

```
ld_spawn(uname,prname,prio,pc,pv,uc,uv,pid,errblock)
  char *uname, *prname;
  int  prio, pc;
  char *pv(..);
  int uc;
  short int uv(..), *pid;
  char *errblock;
```

This subroutine spawns a process from a loaded program.

The parameters are identical to those of the ld_prod routine.

6.6.3. Spawn Process from In-memory Program.

```
im_spawn(start,prname,prio,pc,pv,uc,uv,pid,stack);
  int (*start) ();
  char *prname;
  int  prio, pc;
  char *pv(..);
  int uc;
  short int uv(..), *pid;
  int stack;
```

This subroutine spawns a process from an in-memory program. The code for the offspring process must be part of the parent process' memory, and execution starts at the indicated address.

The offspring process will share the global but not the local variables of the parent process.

Parameters:

Most parameters are identical to those of the ld_prod subroutine.

start   is the address of the subroutine to be started as a process.

stack   is the number of bytes to be reserved for the offspring process stack.

6.6.4. Produce Process from Installed Program.

```
in_prod(iname,prname,prio,pc,pv,uc,uv,pid)
  char *iname, *prname;
  int  prio, pc;
  char *pv(..);
  int uc;
  short int uv(..), *pid;
```

This subroutine produces a new process from an installed program.

Parameters:

Most parameters are identical to those of the ld_prod routine.

iname is the address of an 8 character array containing the name of the installed program.

prio is the process priority.

6.6.5. Spawn Process from Installed Program.

```
in_spawn(iname,prname,prio,pc,pv,uc,uv,pid)
  char *iname, *prname;
  int  prio, pc;
  char *pv(..);
  int uc;
  short int uv(..), *pid;
```

This subroutine spawns a new process from an installed program.

The parameters are identical to those of the in_prod routine.

6.6.6. Process Metamorphosis from Loaded Program.

```
ld_meta(uname,pc,pv,errblock)
  char *uname;
  int pc;
  char *pv(..), *errblock;
```

This subroutine causes the calling process to metamorfose. The code for the new program is loaded.

Parameters:

uname is the address of a null-terminated string being the unitname of the file containing the program.

pc is the number of parameter strings passed to the metamorphosed process.

pv is the address of an array of null-terminated parameter strings to be passed to the metamorphosed process.

errblock is the address of a 6-byte array in which additional error information may be stored by the disk driver if a hard disk error occurs during program load.

6.6.7. Process Metamorphosis from Installed Program.

```
in_meta(iname,pc,pv)
  char *iname;
  int pc;
  char *pv(..);
```

This subroutine causes the calling process to metamorfose. The code for the new program is installed.

Parameters:

iname is the address of an 8 character array containing the name of the installed program.

pc is the number of parameter strings passed to the metamorphosed process.

pv is the address of an array of null-terminated parameter strings to be passed to the metamorphosed process.

6.6.8. Fork.

```
prc_fork(prname,pid)
  char *prname;
  short int *pid;
```

This subroutine causes the calling process to fork.

Parameter:

prname is the address of an 8 character string containing the name which is to be assigned to the offspring process. If prname==0, Unirex assigns the name ´$$$$nnnn´ to the process, where nnnn is the hexadecimal value of the process number.

pid is the address of the location where the fork information will be stored. In the parent process, the process number of the offspring process will be stored. In the offspring process, 0 will be stored.

6.6.9. Install Program.

```
ins_prog(uname,iname,errblock)
  char *uname, *iname;
  char *errblock;
```

This subroutine installs a program. This subroutine may be called by privileged processes only.

Parameters:

uname the address of the null-terminated unitname of the file containing the program.

iname the address of an 8 character string containing the name to be assigned to the installed program.

errblock the address of a 6-byte array in which error information from the disk drive may be stored.

6.6.10. Remove Installed Program.

```
rem_prog(iname)
  char *iname;
```

This subroutine removes an installed program. This subroutine may only be called by privileged processes.

Parameter:

iname the address of an 8 character string containing the name of the installed program.

6.6.11. Exit.

```
exit(cc)
  short int cc;
```

This routine causes the calling process to die. The completion code, cc, is reported to parent process, if any.

6.6.12. Abort Process.

```
abo_prc(pid,cc)
  short in pid, cc;
```

This routine aborts a process. Unprivileged users may only abort processes belonging to themselves.

Parameters:

pid — is the number of the process to be aborted.

cc — is the condition code to be reported to the parent process, if any, of the killed process.

6.6.15. Suspend Process.

```
susp_prc(pid,time)
  short int pid;
  int time;
```

This directive suspends the execution of a process. The process issuing the call is itself suspended if pid==0. Unprivileged users may only suspend processes belonging to themselves.

The execution of the process is resumed upon the calling (by another process) of rsum_prc or the expiration of the specified time, whichever comes first.

If a process has suspended itself, susp_prc will return zero if execution was resumed because of an expired time. The value ERESUME will be returned if execution was resumed becaus of a rsum_prc call issued by another process.

Parameters:

pid — is the number of the process to be suspended. The calling process is itself suspended if pid==0.

time — is the duration of the suspension in centiseconds. Execution is suspended indefinately if time==0.

6.6.14. Resume Process.

```
rsum_prc(pid)
  short int pid;
```

This routine resumes the execution of a suspended process. Unprivileged users may only resume processes belonging to themselves.

Parameter:

pid   is the number of the process to be resumed.

6.6.15. Get Process Status.

```
prc_stat(pid,block)
  short int pid;
  struct
    begin
      char      name (.8.);
      short int ruser, euser;
      int       susptime;
      short int prio, asn;
      struct
        begin
          int       physadd, length;
          short int rw;
        end     memory (.16.);
      short int subproc, kind;
      int       priv:1, act:1, runn:1, susp:1, wait:1, abo:1;
    end  *block;
```

This routine fetches information about a process.

Parameters:

pid — is the number of the process. The calling process is itself assumed if pid==0.

block — is the address of a memory location where the process information will be stored in the following format:

block->name	will contain the name of the process.
block->ruser	will contain the real user number of the process.
block->euser	will contain the effective user number of the process.
block->susptime	will contain the number of centiseconds that the process will yet be suspended.
block->prio	will contain the process priority.
block->asn	will contain the process asn.
block->memory(.i.).physadd	will contain the pysical address of process memory segment i.
block->memory(.i.).length	will contain the length of process memory segment i.
block->memory(.i.).rw	will be 0 if the process has no access to segment i, 1 if the process

has read-only access, 3 if the process has read/write access.

block->subproc — will contain the number of sub-processes subordinate to this process.

block->kind — will contain

- KINDLMP for a loaded main process,
- KINDLSP for a loaded sub-process,
- KINDIMP for an installed main process,
- KINDISP for an installed sub-process,
- KINDMEM for an in-memory sub-process.

block->priv — will be one if the process i privileged.

block->act — will be one if the process i currently active.

block->runn — will be one if the process is currently running.

block->susp — will be one if the process is currently suspended.

block->wait — will be one if the process is waiting for I/O.

block->abo — will be one if the process i being aborted.

6.6.16. Get Process Number.

```
proc_num(pname,user,pid)
  char *pname;
  short int user, *pid;
```

This routines fetches the process number of a process.

Parameters:

pname is the address of an 8 character string containing the name of the process. If pname==0, the calling process is assumed.

user is the number of the user owning the process. If user==0, the real user number of the calling process is assumed.

pid is the address where the process number should be stored.

6.6.17. Change Process Priority.

```
ch_prio(pid,prio)
  short int pid, prio;
```

This routine changes the priority of a process. Unprivileged processes may only change the priority of processes with the same real user number as the calling process.

Parameters:

pid   is the process number. If pid==0, the calling process is assumed.

prio   is the new priority.

6.6.18

Documentation for subroutines to handle exeptions and terminal attentions (UNIX ´signals´) will be added at a later state.

7. Memory.

A user process has access to up to 14 megabyte of memory. Normally, one or more readonly segments and one or more read/write segments are allocated to the user process. The user may desire to allocate additional memory during program execution, or the user may wish to access memory allocated by another process. This section describes how this is done.

7.1. Partitions.

Data areas allocated during program execution are termed ´partitions´. A process may create a partition, and, optionally, allow other processes to access this partition.

One special use of partitions is for resident subroutine libraries. Partitions available to all users may be created, and these partitions may, for example, contain often-used subroutines. The user programs may access these subroutines simply by ´attaching´ (see below) to the appropriate partition.

Before a process may use a partition the process must be ´attached´ to that partition. A partition may or may not be deleted when no process is attached to it.

Processes attaching to a partition will map certain logical addresses onto the physical addresses of the partition, using one segment per partition. Thus the maximum of 14 segments accessible to each process sets a limit to the number of partitions to which a process can be attached at any given time.

7.2. Operations on Partitions.

7.2.1. Creating a Partition.

A user process may create a partition. The partition is given a name and the process is automatically attached to it. A certain memory segment (certain logical addresses) are mapped to the physical addresses of the partition. The partition is said to belong to the user issuing the request. When a process creates a partition, it specifies the access rights of other processes to the partition.

7.2.2. Attaching to a Partition.

Before a process can access a partition, the process must be attached to that partition. When a process is attached to a partition, a certain memory segment (certain logical addresses) are mapped to the physical addresses of the partition.

7.3. System Calls.

The following system calls exist:

7.3.1. Create Partition.

```
crea_par(pname,length,laddr,access,delete)
  char *pname;
  int length, laddr;
  short int access;
  char delete;
```

This subroutine creates a partition and assigns it a name. This partition will belong to the user, with the real user number of the calling process. The calling process is automatically attached to the partition. The calling process is allowed read/write access to the partition. Other processes may request read/write or read-only access depending on the value of the access parameter.

Parameters:

pname is the address of an 8-character string specifying the name of the partition. This name must be unique for the user, but not necessarily unique within the system.

length is the length in bytes of the partition. This number must be at most 0x1000000, and will be rounded to the next hight number divisible by 0x100.

laddr is the logical address to which the partition should be mapped. Only bits 20-23, specifying the segment number, are used.

access is the specification of what access rights of other processes to the partition. This number is in the same format as the protection specification for I/O units (see section 8.x).

delete specifies if the partition should be automatically deleted when no process is attached to it. If delete==1 automatic deletion takes place.

7.3.2. Delete Partition.

```
del_par(pname,user)
  char *pname;
  short int user;
```

This subroutine deletes a partition provided that the process has access to the partition. If a process is attached to the partition, deletion is postponed until all processes have detached from the partition.

Parameters:

pname is the address of an 8 character string specifying the name of the partition.

user is the number of the user owning the partition. If user==0 the real user number of the calling proces is used.

7.3.3. Attach Partition.

```
att_par(pname,user,laddr,access)
  char *pname;
  short int user;
  int laddr;
  char access;
```

This subroutine attaches the process issuing the call to a partition.

Parameters:

pname — is the address of an 8 character string specifying the name of the partition.

user — is the number of the user owning the partition. If user==0 the real user number of the calling proces is used.

laddr — is the logical address to which the partition should be mapped. Only bits 20-23, specifying the segment number, are used.

access — specifies the requested access. access==1 means read-only access, access==3 means read/write access.

7.3.4. Detach Partition.

```
det_par(pname,user)
  char *pname;
  short int user;
```

This subroutine detaches the calling process from a partition. The process must be attached to the partition when the call is made.

Parameters:

pname   is the address of an 8 character string specifying the name of the partition.

user   is the number of the user owning the partition. If user==0 the real user number of the calling proces is used.

7.3.5. Get Partition Status.

```
par_stat(pname,user,block)
  char *pname;
  short int user;
  struct begin
    int length;
    short int access;
    int paddr;
    short int att;
  end *block;
```

This subroutine gets status information about a partition.

Parameters:

pname is the address of an 8 character string specifying the name of the partition.

user is the number of the user owning the partition. If user==0 the real user number of the calling proces is used.

block is the address of a memory location where the status information should be stored. Upon return from the call

- block->length will contain the length of the partition.
- block->access will contain the access specification for the partition.
- block->paddr will contain the physical address of the partition.
- block->att will contain the number of processes attached to the partition.

7.3.6. Get Memory Information.

```
mem_info(paddr,block)
  int paddr;
  struct begin
    short int type;
    char      name(.8.);
    short int user;
    int       length;
    short int access;
    int       padd;
  end *block;
```

This directive gets information about the memory usage in the computer.

Parameters:

paddr specifies a physical address. The information returned will be about the usage of the next memory partition at a physical address greater than paddr.

block is the address of the memory location where information about the next memory partition (including normal program memory) should be stored. Upon return from the call

block->type	will contain the partition type.
block->name	will contain the partition name.
block->user	will contain the partition owner number.
block->length	will contain the length of the partition.
block->access	will contain the access right information of the partition.
block->padd	will contain the physical address of the paratition (this number may be used in a subsequent mem_info call).

8. I/O Management.

Unirex handles input and output in a manner that is, as far as possible, device independent. All I/O is performed on an ´I/O unit´ which may be

1) The ´null device´.
2) A terminal.
3) A printer.
4) A disk.
5) A Mikfile file on a disk.
6) A Unifile file on a disk.
7) A box.
8) A system box.
9) A common box. (Not implemented in the first release of Unirex.)

An I/O unit may reside on another computer linked to ´our´ computer through the Uninet. A full I/O unit specification takes the form:

!computer:device/name/name/name

computer is the name of the computer on which the I/O resides. If !computer is omitted, ´this´ computer is assumed.

device is the name of the device on which the I/O unit resides. This specification may be:

1) :null for the null device.
2) :term01, :term02, etc. for terminal number 1, 2, etc.
3) :print01, :print02, etc. for printer number 1, 2, etc.
4) :disk01, :disk02, etc. for disk number 1,2, etc. and for files residing on those disks.
5) :box for boxes.
6) :sysbox for system boxes.
7) :combox for common boxes.

If :device is omitted, ´:disk01´ is assumed.

/name/name/name is the specification of a file on a disk or the name of a box, system box, or common box.

In the system calls, unit specifications (the so-called ´unit names´) are always specified as a 0-terminated character string.

All characters in the unit names are ideally lower case letters. Upper case letters are converted to the lower case counterparts.

If the first character of a unit name is neither ! nor : nor /, the unit name is prefixed by the so-called ´current unit prefix´ which is a property of a process. If, for example, the current unit prefix of a process is ´:disk02/alpha/beta/´, and the process specifies unit name ´gamma/delta´, the effective unit name will be ´:disk02/alpha/beta/gamma/delta´.

When a unit is opened or created it is assigned an I/O unit descriptor which is a short integer that should be used in subsequent operations on the file. The value of the I/O unit descriptor is always the smallest value currently not assigned to an open I/O unit.

8.1. The Devices.

8.1.1. The Null Device.

The Null Device is used for disposing of unwanted output. On output the null device is a bottomless pit, on input it always yields an end-of-file. The specification of this device is

:null

8.1.2. Terminals and Printers.

I/O is identical on terminals and printers. The only difference is that the opening of an I/O unit being a printer involves the reservation of that printer, whereas this is not the case with terminals.

Terminals and printers are numbered 1, 2, etc.

The specification of terminal number 5 is

:term05

The specification of printer number 5 is

:print05

8.1.3. Disks.

I/O to a disk may be either direct or via a file system. For direct I/O the reading and writing of byte strings is supported. The disks are numbered 1, 2, etc. whith disk number 1 being the default disk used when no device specification is present. For direct disk I/O the specification of disk number 3 is

:disk03

8.1.4. Mikfile.

Mikfile is the MIKADOS-compatible file system. Files under Mikfile have names comprised of up to 8 characters, followed by a period, followed by one character. The specification of the Mikfile file hanoi of type k on disk 4 is

:disk04/hanoi.k

8.1.5. Unifile.

Unifile is the UNIX V/7 compatible file system. Files under Unifile have names comprised of up to 14 characters, possibly followed by another file name specification. The specification of the Unifile file hanoi residing in the directory alpha, which resides in the directory beta on disk 2 is

:disk02/beta/alpha/hanoi

A special case of Unifile files are the so-called redirection files. A redirection file contains a unit name, which replaces the part of the unit name used to reach the redirection file. If, for example, the file :disk01/pip/pop is a redirection file, containing the unit name ´:disk03/first/second´, the unit name ´/pip/pop/alpha/beta´ (using :disk01 by default) is effectively the unit name ´:disk03/first/second/alpha/beta´.

Or, if :disk01/dev/tty1 is a redirection file, containing ´:term01´, the unit name ´/dev/tty1´ is effectively the unit name ´:term01´.

8.1.6. Boxes and System Boxes.

Boxes are used for message exchange and synchonization between processes. A box is logically an I/O unit, but is resident within a master CPU. It contains a buffer into which data may be written and read. Before a process may use a box the process must open it, or, perhaps, create it if it did not already exist. A box is automatically deleted when it is no longer open for any process and contains no data, except for the so-called system boxes which must be deleted by an explicit call to the delete routine. Only privileged processes may create and delete system boxes.

Access protection applies to boxes in a manner analogous to files, however opening a box is always in read/write mode with no reservation of the box.

Boxes have names of up to 8 characters.

The specification of the box ´frodo´ is

:box/frodo

The specification of the system box ´gandalf´ is

:sysbox/gandalf

8.1.7. Common Boxes.

Common boxes are system boxes located in the memory common to all CPUs in the Unimat.

The specification of the common box ´bilbo´ is

:combox/bilbo

8.2. I/O Unit Protection.

All I/O units are protected by a protection mode specifier of 4 bytes. Two bytes contain the number of the user owning the I/O unit, and the low order 12 bits of the other two bytes specify the access rights in the form ugtrwxrwxrwx. This is identical to the UNIX protection, except that the t-bit is always ignored in Unirex, and the u-bit is considered on if the g-bit is on.

When Unirex is loaded the following protection right apply to the various devices:

Device	Owner	Protection	
:null	0	---rw-rw-rw-	
:termNN	0	---rw-rw-rw-	
:printNN	0	---rw-rw-rw-	
:disk01	1 (Unifile)	---rw-rw----	
:diskNN	0	---rw-rw-rw-	(all disks exept :disk01)

The computer startup command file may specify a change in these access rights.

8.3. File Systems.

The two file systems Mikfile and Unifile are supplied with the Unirex system. The user may himself create other file systems.

A file system is a process satisfying certain requirements as specified in section 8.x. When a disk is mounted it is specified which file system process should operate on the disk.

It is the duty of the file system to convert the various file I/O requests into relevant input and output operations directly on the disk on which the file system works.

8.4. System Calls.

The following sections list the various I/O system calls. For each call a short description is given, followed by a parameter list, followed by a description of how the call works on particular units.

8.4.1. Create Unit.

```
creat_un(ioud,uname,prot,mode,size,errblock)
  short int *ioud;
  char *uname;
  short int prot, mode;
  int size;
  char *errblock;
```

This subroutine creates an I/O unit.

Parameters:

ioud — is the address of the location where the I/O unit descriptor should be stored.

uname — is the address of the null-terminated unit name.

prot — is the protection bits of the created unit.

mode — specifies the access mode of the unit. mode==2 for write, mode==3 for read/write, mode==4 for selective update.

size — is the size of the unit.

errblock — is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null and :termNN

prot, mode, size, and errblock are ignored. mode is always assumed to be 3.

:printNN

prot, mode, size, and errblock are ignored. mode is always assumed to be 3. The printer is reserved so that no other process may open/create it.

:diskNN

Not allowed.

Mikfile

If the file does not exist, it is created. If it does exist, the call fails. mode is always assumed to be 3. The file is reserved so that no other process may open it. size specifies the size of the file in 256 byte sectors.

Unifile

If the file does not exist, it is created. If it does exist, it is truncated to zero length. The file is reserved as required by mode. size specifies the number of contiguous 256 (512?) byte sectors that are to be reserved for the file. size may be 0. Specifying a size does not alter the functioning of the file, but it may improve the access time.

:box/nnnnnnnn

If the box does not exist, it is created. If it does exist, the call fails. mode and errblock are ignored. mode is always assumed to be 3. size specifies the number of bytes in the box buffer.

:sysbox/nnnnnnnn

If the box does not exist, it is created. If it does exist, the call fails. mode and errblock are ignored. mode is always assumed to be 3. size specifies the number of bytes in the box buffer. Only privileged processes may perform this call.

8.4.2. Create Extent to Mikfile.

```
crext_un(ioud,errblock)
  short int ioud;
  char *errblock;
```

This subroutine creates an extent to a Mikfile file.

Parameters:

ioud   is the I/O unit descriptor of the file.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

Mikfile.

This call will extent the specified file, provided it has less than 60 extents.

All other units.

Not allowed.

8.4.3. Open Unit.

```
open_un(ioud,uname,mode,errblock)
  short int *ioud;
  char *uname;
  short int mode;
  char *errblock;
```

This subroutine opens an I/O unit.

Parameters:

ioud — is the address of the location where the I/O unit descriptor should be stored.

uname — is the address of the null-terminated unit name.

mode — specifies the access mode of the unit. mode==1 for read, mode==2 for write, mode==3 for read/write, mode==4 for selective update.

errblock — is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null and :termNN

mode and errblock are ignored. mode is always assumed to be 3.

:printNN

mode and errblock are ignored. mode is always assumed to be 3. The printer is reserved so that no other process may open/create it.

:diskNN

errblock is ignored.

Mikfile

The specified file is opened, if it exists. Only mode==1 and mode==3 are allowed. The file is reserved as required by mode.

Unifile

The specified file is opened, if it exists. The file is reserved as required by mode.

:box/nnnnnnnn and :sysbox/nnnnnnnn

The specified box is opened, if it exists. mode and errblock are ignored. mode is always assumed to be 3.

8.4.4. Close Unit.

```
close_un(ioud,errblock)
  short int ioud;
  char *errblock;
```

This subroutine closes an I/O unit.

Parameters:

ioud is the I/O unit descriptor.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null and :termNN

errblock is ignored.

:printNN

errblock is ignored. The printer reservation is released if this was the last close for this printer.

:diskNN

errblock is ignored.

Mikfile and Unifile

The file reservation is released if this was the last close for this file.

:box/nnnnnnnn

errblock is ignored. The box is deleted if this was the last close for this box, and the box buffer is empty.

:sysbox/nnnnnnnn

errblock is ignored.

8.4.5. Get Data from Unit.

```
get_un(ioud,buf,count,actual,errblock)
  short int ioud;
  char *buf;
  int count, *actual;
  char *errblock;
```

This subroutine reads from an I/O unit.

Parameters:

ioud — is the I/O unit descriptor.

buf — is the address of the location where the data input should be stored.

count — is the size of the input buffer. No more than this number is input.

actual — is the address of the location where the actual number of characters input should be stored.

errblock — is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

buf, count, and errblock are ignored. *actual is always set to zero.

:termNN and :printNN

errblock is ignored.

If the device operates in line mode, a line of up to count-1 characters is input from the terminal. This line is stored in buf with a trailing lf character (ASCII code 0x0a). *actual is set to the number of characters input, including the lf. If the eof key is pressed, *actual is set to 0.

If the device operates in direct input mode, the number of characters input since the last get_un operation are transferred. However, no more than count characters are transferred.

:diskNN and Mikfile and Unifile.

Up to count bytes are input from the unit. If the end of the unit is reached, only the available number of bytes are input.

:box/nnnnnnnn and :sysbox/nnnnnnnn

Up to count bytes are input from the box. If the box contains fewer than count bytes, only the available number of bytes are returned. If the box is expty, the calling process waits until something is written into the box. *actual is thus never set to zero.

8.4.6. Put Data to Unit.

```
put_un(ioud,buf,count,errblock)
  short int ioud;
  char *buf;
  int count;
  char *errblock;
```

This subroutine writes count bytes to an I/O unit.

Parameters:

ioud is the I/O unit descriptor.

buf is the address of the location where the data to be output is stored.

count is the size of the output buffer.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

buf, count, and errblock are ignored.

:termNN and :printNN

errblock is ignored. If the device is in control-sequence mode, any initial <...>-sequence will be interpreted, rather than output. If the device is in Mikados mode, the output will be terminated by a lf/cr, unless a <S> is in effect. If the device in UNIX mode, any lf in the buffer will be output as lf/cr.

:diskNN and Mikfile

If the end of the unit will be reaced before output terminates, nothing is output.

Unifile

The file is extended as required.

:box/nnnnnnnn and :sysbox/nnnnnnnn

If the box is too full to contain count bytes, the outputting process waits until the box is sufficiently empty. An error occurs if the size of the output buffer exceed the total box buffer size.

8.4.7. Get Data Backwards from Unit.

```
getb_un(ioud,buf,count,actual,errblock)
  short int ioud;
  char *buf;
  int count, *actual;
  char *errblock;
```

This subroutine reads backwards from an I/O unit.

Parameters:

ioud — is the I/O unit descriptor.

buf — is the address of the location where the data input should be stored.

count — is the size of the input buffer. No more than this number is input.

actual — is the address of the location where the actual number of characters input should be stored.

errblock — is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

buf, count, and errblock are ignored. *actual is always set to zero.

:termNN and :printNN

Not allowed.

:diskNN and Mikfile and Unifile.

Up to count bytes are input from the unit. If the beginning of the unit is reached, only the available number of bytes are input.

:box/nnnnnnnn and :sysbox/nnnnnnnn

Not allowed.

8.4.8. Update Buffer on Unit.

```
edit_un(ioud,buf,count,actual,notmod,curoff,errblock)
  short int ioud;
  char *buf;
  int count, *actual;
  short int notmod, curoff;
  char *errblock;
```

This subroutine outputs the buffer, allows the user to change it, and inputs it again, provided the I/O unit is a terminal. On other devices, this is identical to get_un.

Parameters:

ioud is the I/O unit descriptor.

buf is the address of the buffer, whose contents are to be altered.

count is the size of the buffer. No more than this number is input.

actual is the address of the location where the actual number of characters input should be stored.

notmot is the number of characters at the beginning of the buffer that should not be output.

curoff is the initial cursor offset when input starts.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

buf, count, curoff, and errblock are ignored. *actual is always set to notmod.

:termNN

errblock is ignored. The terminal must operate in line mode. If the device is in control-sequence mode, any initial <...>-sequence will be interpreted, rather than output. The buffer should not contain any control characters except, perhaps a final lf. Anyway, the final character of the buffer is always ignored. A trailing lf will be stored in the buffer. *actual is set to the number of characters input, including the lf. If the eof key is pressed, *actual is set to notmod.

All other devices.

edit_un works as get_un to a buffer with the length count-notmod, and the first notmod characters of buf are not changed.

8.4.9. Position Unit.

```
pos_un(ioud,count,mode,errblock)
  short int ioud;
  int count;
  short int mode;
  char *errblock;
```

This subroutine positions an I/O unit to a particular byte.

Parameters:

ioud is the I/O unit descriptor.

count is the desired unit position.

mode controls the interpretation of count. If mode==0, count is absolute counting from the beginning of the unit. If mode==1, count is added to the current unit posistion. If mode==2, count is absolute counting backwards from the end of the unit.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

The call is ignored.

:termNN and :printNN

Not allowed.

:diskNN

errblock is ignored.

Mikfile

mode must not be 2.

Unifile

Works ok.

:box/nnnnnnnn and :sysbox/nnnnnnnn

Not allowed.

8.4.10. Link a File to a Directory.

```
link_dir(old,new,errblock)
  char *old, *new, *errblock;
```

This subroutine creates a link in a directory to a file.

Parameters:

old — is the address of the null-terminated unit name of the file.

new — is the address of the null-terminated new name by which the file should be known.

errblock — is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

Unifile.

Works ok.

All other units.

Not allowed.

8.4.11. Unlink a File from a Directory.

```
unl_dir(uname,errblock)
  char *uname, *errblock;
```

This subroutine removes a link in a directory to a file.

Parameters:

uname is the address of the null-terminated unit name of the file.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

Unifile.

If the last link to a file is removed, the file is itself removed. If, however, the file is in use by some other process, the deletion of the file is postponed until it is closed.

All other units.

Not allowed.

8.4.12. Rename a File.

```
ren_file(old,new,errblock)
  char *old, *new, *errblock;
```

This subroutine renames a file.

Parameters:

old — is the address of the null-terminated unit name of the file.

new — is the address of the null-terminated new name by which the file should be known.

errblock — is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

Unifile.

The call is implemented as

```
        link_dir(old,new,errblock);
        unl_dir(old,errblock);
```

Mikfile.

Works ok.

All other units.

Not allowed.

8.4.13. Delete a Unit.

```
del_un(uname,errblock)
  char *uname, *errblock;
```

This routine deletes an I/O unit.

Parameters:

uname is the address of the null-terminated unit name.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null and :termNN and :printNN and :diskNN

Not allowed.

Mikfile

The file must not be open when the call is made.

Unifile

The call is implemented as unl_dir(uname,errblock).

:box/nnnnnnnn

errblock is ignored. The box is deleted even if it is not empty. If som process has the box open, deletion will be postponed until the box is closed. Note that empty boxes are automatically deleted when closed.

:sysbox/nnnnnnnn

errblock is ignored. The system box is deleted even if it is not empty. If som process has the box open, deletion will be postponed until the box is closed. Only privileged processes may perform this call.

8.4.14. Change Unit Access Mode.

```
chacc_un(uname,mode,errblock)
  char *uname;
  short int mode;
  char *errblock;
```

This call changes the access right bits of an I/O unit.

Parameters:

uname is the null-terminated name of the unit.

mode is the new access mode. The low order 12 bits specify ugtrwxrwxrwx (as in UNIX).

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

The call is ignored.

All other units.

Works ok.

8.4.15. Get Load Module Information.

```
info_lm(uname,block,errblock)
  char *uname;
  struct begin
    struct begin
      char rw;
      int  length;
    end segment(.16.);
    int start, stackb, stacke;
    char setunum;
    short int user;
    int serial;
  end *block;
  char *errblock;
```

This subroutine gets load module information. This subroutine may only be called by privileged processes.

Parameters:

uname is the null-terminated name of the file containing the load module.

block is the address of the memory location where the load module information should be stored.

block->segment(.i.).rw	will be 1 if segment i is a read/write segment, 0 if segment i is a read-only segment.
block->segment(.i.).length	will be the length of segment i.
block->start	will be the execution start address of the program.
block->stackb	will be the lowest address of the stack area.
block->stacke	will be the address of the byte following the stack area.
block->setnum	will be the value of the u-bit logically or´ed with the g-bit of the load module.
block->user	will be the owner of the load module.
block->serial	will be the encrypted serial number of the computer on which the program may run.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

Mikfile and Unifile

Works ok.

All other units

Not allowed.

8.4.16. Load Load Module.

```
load_lm(uname,addr,errblock)
  char *uname;
  int addr(.16.);
  char *errblock;
```

This routine loads a load module into memory. The routine is not available to the user, but must be supported by the file systems.

Parameters:

uname   is the address of the null-terminated load module file name.

addr   is the address of 16 physical (!) memory addresses into which the program segments should be loaded.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

Mikfile and Unifile

Works ok.

All other units

Not allowed.

8.4.17. Get Unit Status.

```
stat_un(uname,block,errblock)
  char *uname;
  struct stat *block;
  char *errblock;
```

This subroutine gets I/O unit status information.

Parameters:

uname — is the null-terminated name of the unit.

block — is the address of the memory location where the status information should be stored. See below for the layout of the structure stat.

errblock — is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

block will be set to 0.

:termNN and :printNN

```
struct stat begin
          short int user,  /* the access right user number */
                    mode;  /* the access right mode bits   */
        end;
```

See also the get_sioc subroutine.

:diskNN

```
struct stat begin
          short int user,  /* the access right user number */
                    mode;  /* the access right mode bits   */
          int sect;        /* the number of sectors on the disk */
          char type;       /* the disk type */
        end;
```

Mikfile and Unifile

The actual layout of struct stat has not been determined.

:box/nnnnnnnn and :sysbox/nnnnnnnn

```
struct stat begin
         short int user,  /* the access right user number */
                   mode;  /* the access right mode bits   */
         int ibytes,      /* the number of bytes that may be
                             read from the box */
             obytes;      /* the number of btyes that may be
                             written to the box */
      end;
```

8.4.18. Generate Box Name.

```
gen_boxn(sys,*uname)
  short int sys;
  char *uname;
```

This routine generates a new unique name of a box. The name will start with two $-signs, and this should thus not be used in user-generated box names.

sys — is 1 if the name of a system box is requested, 0 for non-system boxes.

uname — is the address of the location where the generated name should be stored. It will be a null-terminated string in the form ´:box/$$nnnnnn´ or ´:sysbox/$$nnnnnn´, where nnnnnn is some number.

8.4.19. Get Terminal or Printer Operation.

```
get_sioc(uname,block)
  char *uname;
  struct siocsta *block;
```

This subroutine gets information about a terminal or printer.

uname is the null-terminated name of the I/O unit.

block is the address of the memory location where the unit information should be stored. The layout of the structure siocsta has not yet been determined.

8.4.20. Set Terminal or Printer Operation.

```
set_sioc(uname,block)
  char *uname;
  struct siocsta *block;
```

This subroutine sets terminal or printer operation.

uname is the null-terminated name of the I/O unit.

block is the address of the memory location where the operation specification is stored. The layout of the structure siocsta has not yet been determined.

8.4.21. Get Function Key Value.

```
get_fkey(ioud,key)
  short int ioud;
  char *key;
```

This subroutine fetches the value of the most resently depressed function key on an I/O unit.

Parameters:

ioud is the I/O unit descriptor.

key is the address of the memory location where the information should be stored.

:termNN and :printNN

Works ok.

All other units

Not allowed.

8.4.22. Duplicate I/O Unit Descriptor.

```
dup_ioud(old,new)
  short int old, *new;
```

This subroutine associates a new I/O unit descriptor with an already open unit.

Parameters:

old   is the old I/O unit descriptor.

new   is the address of the memory location where the new I/O unit descriptor should be stored. The I/O unit referenced by *new will in all respects be identical to the I/O unit referenced by old.

8.4.25. Copy I/O Unit Descriptor.

```
cop_ioud(old,new)
  short int old, new;
```

This subroutine copies an old I/O unit descriptor into a new one.

Parameters:

old   is the old I/O unit descriptor.

new   is the new I/O unit descriptor. The I/O unit referenced by new will in all respects be identical to the I/O unit referenced by old. new must not be associated with an open I/O unit when the call is made.

8.4.24. Get Current Unit Prefix.

```
get_cup(*pname)
  char *pname;
```

This subroutine fetches the current unit prefix for the calling process.

Parameter:

pname is the address of the memory location where the null-terminated current unit prefix will be stored.

8.4.25. Set Current Unit Prefix.

```
set_cup(*pname)
  char *pname;
```

This subroutine sets the current unit prefix for the calling process.

Parameter:

pname   is the address of the null-terminated new current unit prefix.

8.4.26. Change the Owner of a Unit.

```
chown_un(uname,owner,errblock)
  char *uname;
  short int owner;
  char *errblock;
```

This call changes the user number determining access rights to an I/O unit. This call may only be performed by privileged processes.

Parameters:

uname   is the null-terminated name of the unit.

owner   is the new unit owner user number.

errblock is the address of a 6 byte memory location where the disk driver may store error information when a hard disk error is encountered.

:null

The call is ignored.

All other units.

Works ok.

8.4.26. Mount a Disk.

```
m_disk(disk,filesys)
  char *disk, *filesys;
```

This subroutine mounts a disk, that is, associates it with a file system process.

Parameters:

disk is the address of the null terminated unit name of the disk.

filesys is the address of the 8 character name of the file system process, normally "mikfile" or "unifile".

8.4.27. Unmount a Disk.

```
um_disk(disk)
  char *disk;
```

This subroutine unmounts a disk, that is, disassociates it with a file system process.

Parameters:

disk   is the address of the null terminated unit name of the disk.

9. Miscellaneous System Services.

9.1. Get System Time.

```
get_time(sec,msec)
  int *sec;
  short int *msec;
```

This subroutine fetches the system time. The system time is measured in seconds since 00:00:00 GMT, January 1, 1970. Using a 4 byte integer to hold the seconds makes this convention useable until the year 2106.

Parameters:

sec is the address of the memory location where the system time in seconds should be stored.

msec is the address of the memory location where the milliseconds counter should be stored.

9.2. Set System Time.

```
set_time(sec)
  int sec;
```

This subroutine sets the system time. The routine may only be called by privileged processes.

Parameter:

sec   is the number of seconds since 00:00:00 GMT, January 1, 1970.

9.3. Inter-Process Move.

```
ip_move(spid,saddr,dpid,daddr,count)
  short int spid;
  int saddr;
  short int dpid;
  int daddr, count;
```

This subroutine moves a block of data from one process to another. This subroutine may only be called by privileged processes.

Parameters:

spid is the process number of the source process. The calling process is assumed if spid==0.

saddr is the logical address of the source block within the source process.

dpid is the process number of the destination process. The calling process is assumed if dpid==0.

daddr is the logical address of the destination block within the destination process.

count is the number of bytes to move.

9.4. Get Hardware Configuration.

```
get_hw(block)
  struct begin
    short int term,
              print,
              disk,
              cpu,
              sioc,
              dioc;
  end *block;
```

This subroutine fetches information about the hardware configuration of the computer.

Parameter:

block   is the address of the memory location where the hardware information should be stored:

```
block->term   will be the number of terminals.
block->print  will be the number of printers.
block->disk   will be the number of disks.
block->cpu    will be the number of master CPUå.
block->sioc   will be the number of SIOCs.
block->dioc   will be the number of DIOCs.
```