Cromemco®

Cromix-Plus™ Programmer's

Reference Manual

Cromemco®

Cromix-Plus™ Programmer's

Reference Manual

023-5014

October 1987

Rev. F

CROMEMCO, Inc.
P.O. Box 7400
280 Bernardo Avenue
Mountain View, CA 94039

This manual was produced using a Cromemco System 300 computer running under the Cromemco UNIX Operating System. The text was edited with the Cromemco CE Editor. The edited text was formatted by the UNIX TROFF formatter and printed on a Texas Instruments OmniLaser 2108 printer.

CONTENTS

Chapter 1 - Introduction to Cromix-Plus System Calls

The object library **/usr/lib/syslib.obj** contains a number of functions that can be called from a C program. Most of the functions are system call interfaces.

The system call consists of the TRAP #0 instruction followed by a word specifying the system call number. To relieve the user from writing assembler code, all system calls are available as functions in **/usr/lib/syslib.obj** library.

To use system call functions, the programmer must "include" any **include** files that define various structures and constants, into his code. The detailed description of every system call function lists the **include** files that might be useful for every system call. Programmers are strongly encouraged to use the **include** files provided in the **/usr/include** directory.

An Example:

A user wants to write a C program which will turn off the echoing of standard input for the duration of typing in a password. His code might be organized as follows:

```
/*
        Example that shows how to turn off echoing
*/

#include <jsysequ.h>
#include <modeequ.h>
#include <syslib.h>

main()
{
        int oldmode1;

        /* Other parts of the program */

        if ((oldmode1 = setmode(STDIN,MD_MODE1,0,ECHO)) >= 0) {
                /* The echoing is now turned off        */
                /* Here is the code to read password */
                /* Restore echoing to previous state */
                setmode(STDIN,MD_MODE1,oldmode1,ECHO);
        }
        else error(STDERR), exit(ERR);
```

```
        /* Other parts of the program */
}
```

Whenever a system call returns an error, the error number is stored in the global integer variable **errno**. The function **error**, which has a channel number as its only argument, will print out the error message. This mechanism can only be used if no system call function is invoked between the call of the system call function which returned an error, and the call of the **error** function.

The following list summarizes the Cromix-Plus system call functions.

1.1 Summary of System Call Functions

alarm	send alarm signal to calling process after a given number of seconds
boot	boot new operating system
caccess	test channel access
cchstat	change the status of an open file
ccromix	general system call
chdup	duplicate a channel
chkdev	verify presence of a device driver
clink	establish an addition link to an open file
close	close an open file
create	create and open a file
cstat	return the status of an open file
cxexit	terminate execution
cxopen	open a file
delete	delete a directory entry
error	report a system call error
exchg	exchange filenames of two open files
exec	execute a program
faccess	test file access
fchstat	change the status of a file
fexec	fork and execute a program
flink	establish a link to file
fshell	fork a Shell process
fstat	return status of a file
getdate	return the date

getdir	return the current directory pathname
getgroup	return the group number
getmode	return the characteristics of a device
getpos	return the file position
getprior	return the priority of calling process
getproc	return the PID of the calling process
gettime	return the time
getuser	return the user id of the calling process
indirect	general system call
kill	send a signal to a process
lock	lock out processes trying to lock the same sequence
makdev	create a device file
makdir	create a new directory
memory	allocate and deallocate memory
mount	enable access to another file system
msgctl	control operation for a message queue
msgget	get message queue identifier
msgrcv	receive a message from message queue
msgsnd	send a message to a message queue
pause	suspend execution until a signal is sent
phys	allow access to address space outside user memory
pipe	create a pipe
ptrace	trace another process
rdbyte	read a byte
rdline	read a line
rdseq	read specified number of bytes
semctl	control operation for a semaphore set
semget	get semaphore identifier
semop	semaphore operation
setdate	change the system date
setdir	change the current directory
setgroup	change the group id

setmode	change the characteristics of a device
setpos	change the file position
setprior	change the priority of the calling process
settime	change the time
setuser	change the user id
shell	execute a Shell process
shmat	attach the shared memory segment
shmctl	control operations for shared memory segment
shmdt	dettach the shared memory segment
shmget	get shared memory segment identifier
signal	set up a process to receive a signal
sleep	sleep a number of seconds
trunc	truncate the file to the current position
uchstat	change status of a process table
unlock	unlock the lock sequence
unmount	disable access to another file system
update	flush system buffers
ustat	return status of a process table
version	return the operating system version number
wait	wait for the termination of a child process
wrbyte	write a byte
wrline	write a line
wrseq	write a specified number of bytes

1.2 Signals

A signal carries messages between processes. There are eight types of signals that can effect three different responses from a process. The programmer can choose any one of three responses to each of seven of the eight types of signals. The **sigkill** signal in all cases, causes a process to be aborted.

1.3 Responses to a Signal

When a process receives a signal, the signal can be handled in one of three ways.

1. **Ignore the signal.**
   The process continues as though no signal had been received.

2. **Abort the process.**
   The operating system terminates the process.
   This is equivalent to the call of the **cxexit** function.

3. **Transfer control.**
   A user program may establish a location to which control may be transferred for each type of signal received.
   After a signal has been received, the **signal** system call must be executed again in order to be able to receive the next signal.

1.4 Types of Signals

The eight types of signals are enumerated below.

1. **sigabort**
   This is the abort signal generated by a CONTROL-C typed at the terminal.
   The mode of the terminal must be set to allow CONTROL-C to function (**abortenable**).

2. **siguser**
   This is the user signal generated by a character typed at the terminal.
   The character that generates this signal is determined and enabled by mode (**sigchar** and **sigenable**).

3. **sigkill**
   This is the kill signal.
   It cannot be ignored or redirected by the user program.
   The kill signal causes the operating system to abort the process immediately.
   The kill signal can only be sent to a process by the initiator of the process or a privileged user.

4. **sigterm**
   This is the terminate signal.
   It is the default type of signal for the **Kill** command of the Shell.

5. **sigalarm**
   This is the alarm signal.
   It is sent by the operating system following an **alarm** system call.

6. **sigpipe**
   This is the pipe signal.
   It is sent by the operating system when a pipe is not being used properly.

7. **sighangup**
   This is a signal sent by the mtty device when the phone hangs up, if the HUPENABLE mode is set.

8. reserved for future use.

1.5 Sources of Signals

Signals may be sent to a process by a user-typed character, the **Kill** command, the **kill** system call, the **alarm** system call, or by a driver.

1.6 Reception of Signals

A process may be set up to receive and process a signal by the **signal** system call. If the signal is not ignored and the process has an unsatisfied request for input or output from a character device such as a terminal or printer, the input or output request is canceled.

A child process may be set up by its parent process to ignore or be aborted by a signal when the parent initiates the child through the **fexec** or **fshell** system call.

Reaction to signals are determined by the values of **sigmask** and **sigvalue** arguments in the system calls:

bit S-1 in sigmask	bit S-1 in sigvalues	Child's reaction to signal S
0	x	same as parent process
1	0	abort
1	1	ignore

If the child is set up to inherit the parent's reactions and the parent process is set up to trap the signal, the child process will still be aborted by the signal. This is because the child process cannot inherit the parent's trap routine.

The **signal** system call function should be used to install trap routines for signal. This particular system call function is not a straightforward assembler interface. The **signal** function will install its own trap function that will call the user's trap function. The trap function is local to the **signal** system call function and will ensure that all registers will be saved and restored.

Processes initiated by the Shell are set up to inherit reactions to all signals from the parent process, except for the **sigabort**, **siguser**, and **sigterm** signals (these are handled separately).

A process which is run as a detached job by the Shell (through the use of the symbol "&" on a command line) is set up by the Shell to ignore **sigabort** and **siguser** and to be aborted by **sigterm**. A process which runs in the foreground (not detached) is set up by the Shell to react the same way as the parent process (except for interactive Shell processes, which are always set up to ignore those three signals). These features allow the user to abort the current process by entering CONTROL-C, while not affecting detached processes and allow implementation of the Shell command **kill 0**. Additional precaution is taken so that the parent process will not be aborted while the child process is still active.

The **kill** system call sends signals to processes. A user may only send a signal to a process which that user initiated. Only a privileged user may send signals to processes initiated by other users. When a signal is sent to process 0, that signal is sent to all processes initiated from the terminal where the user who invoked the call logged on. If a privileged user sends **siguser** to process 1, system shutdown is

initiated. When **sigabort** is sent to process 1, the Cromix system consults the **/etc/ttys** file to log on any terminals that have been enabled and log off any disabled terminals.

1.7 The Use of Signals in Application Programs

The **signal** system call is commonly used to catch or ignore CONTROL-C (**sigabort**) or other signals.

Immediately after a signal is received, the process is automatically set up to ignore further signals of the same type until the **signal** system call is repeated.

If address 0 is given as the address of the trap routine, the user program will abort on reception of the signal. If address 1 is given as the address of the trap routine, the signal will be ignored.

Signals have many uses, but they also have limitations. Signals are designed to terminate processes or wake them up. Signals are not interrupts. Signals can be ignored, but not disabled. Mutual exclusion cannot be easily achieved with signals. If an application requires that a process receive and process several signals per second from one or more processes, difficulties with stack overflow are likely to arise.

The following program is an illustration that catches the **sigabort** signal sent by the CONTROL-C entered from the terminal.

```
/*
        This program demonstrates the use of the
        signal system call. Note that this program
        will run forever. It cannot be killed by CONTROL-C.
        It must be killed from another terminal.
*/

#include <jsysequ.h>
#include <syslib.h>


main()
{
        setuptrap();            /*Set up trap routine */
        for (;;) ;                /* Infinite loop          */
}


setuptrap()
{
        extern trap_routine();

        if (signal(sigabort,trap_routine) < 0)
                error(STDERR), exit(ERR);
}
```

```
trap_routine()
{
        printf("I do not want to die\n");
        setuptrap();
}
```

1.8 Signals and Forking a New Process

Whenever the user forks a new process which does not fiddle with signals, the forking can be quite simple: the child process should simply inherit signal treatment from the parent process. In more complex cases, there is one pitfall that has to be avoided. It should never happen that the parent process gets killed while the child process is still alive. If this happens, the grandparent process, which is most likely an interactive Shell, will wake up and fight his grandchild process over the characters being input from the terminal. Under such circumstances, the user can never tell which process is going to pick up characters typed on the terminal.

If the child process can set up its own response to signals (it is certainly able to do so if it is an interactive Shell) the parent process must be much more careful. A simple solution is for the parent process, before forking the child process, to set itself up to ignore all signals, storing the old reactions. After the child terminates, the parent process can restore the reactions to their original state. This solution is not always satisfactory: if the user presses CONTROL-C while the child process is running, the parent process will ignore it, though the user might have intended to kill both processes.

A reasonably complete solution can be described as follows:

1. Set up to ignore all signals, storing the old reactions.
2. Inspect the old reactions. If an old reaction was to ignore the signal, keep it that way. If an old reaction was to abort or to trap the signal, a new trap is to be installed. The new trap function (one for each signal) should only note the fact that it was called.
3. Fork a new process with whatever signal reactions are desired, and wait until it terminates.
4. Restore the old signal reactions.
5. If a signal was received in the interim, send the same signal to yourself, thereby causing the same effect (except for the fact that it is postponed).

This description is still not complete, as it does not say what should happen if more than one signal is received in the meantime. This can be handled by the new trap functions and by the processing after the child process terminates. New trap functions can simply set a bit in a word initialized to zero and not establish the trap again. If so, at the end we have a list of signals received while the child was running. The program can now decide which signal to send to itself and in what order (if there is more than one).

1.9 The Alarm System Call

After a specified number of seconds, the **alarm** system call sends an alarm signal (**sigalarm**) to the process that made the system call. The **signal** system call is first used to set up the process for

receiving the **sigalarm** signal. A typical use of **alarm** provides a time out feature for a program. If a process must be prevented from hanging on an input request indefinitely, the process first makes the **alarm** system call. The **alarm** system call specifies the number of seconds to wait after making the request for input.

1.10 The Pause System Call

The **pause** system call is frequently used in conjunction with the **alarm** system call. The **pause** call suspends execution of the calling process and waits for a signal. The **pause** call does not require the **signal** system call to set up the process to receive the signal. It is ideal for putting a process to sleep until another process signals it to continue. The **pause** and **alarm** calls can be used together to put a process to sleep for a specified number of seconds.

For example:

```
if (alarm(10)) error(STDERR);
else pause();
```

1.11 The Sleep System Call

The equivalent of the routine above can be achieved with one system call, **sleep**. The **sleep** call stops execution of a process for a specified number of seconds. The result shown above can be accomplished as follows using **sleep**:

```
sleep(10);
```

1.12 Locks

The **lock** system call assists in implementing file locks, and allows the operating system to absorb part of the overhead involved in the procedure. No locks are imposed by the operating system; this is done by the application program. The **lock** and **unlock** calls merely make and delete entries in a table residing in system memory.

The **lock** system call enters a string in the lock table. This string is the unique identifier of a record in a file. The string is hereinafter referred to as the **lock sequence**. Should another process make a **lock** system call using a lock sequence currently in the lock table, the Cromix Operating System does one of two things. It either puts the process to sleep until the entry is removed, or it returns with an error code set. An entry is removed from the table when the process that made the original **lock** system call reverses it with an **unlock** system call, using the same lock sequence. Any process put to sleep while attempting to lock that sequence is awakened and allowed to make an entry in the table.

The problem of record level lock is resolved by preceding any read or write to a file or record with a **lock** system call. This achieves mutual exclusion for records and avoids the undesirable effects of having multiple processes reading and writing the same file or record.

The other considerations associated with the **lock** system call are the type of lock to be made and the character string to be used as the lock sequence.

1.13 Shared and Unshared Locks

A shared lock allows other processes access to the lock. Shared locks are typically used when a file is being read. A shared lock does not prevent other processes from entering the file, so that a process that is reading a record does not prevent another process from reading the file. A process attempting to establish an unshared lock when a shared lock has been granted either is put to sleep or receives an error.

Unshared locks are typically used during a write to a file, since they prevent any other process from getting access to the lock sequence. If a process has an unshared lock, any other process attempting to lock the same sequence either is put to sleep or receives an error.

1.14 Conditional and Unconditional Locks

Locks can be made conditional or unconditional. A conditional lock returns with an error code set if the sequence specified cannot be locked. An unconditional lock puts the calling process to sleep if the sequence is currently locked. The process put to sleep awakens when the process that originally issued the **lock** call issues an **unlock** call.

The programmer must decide whether to use a conditional or unconditional lock. For many applications, putting a process to sleep for a brief period because another process has locked a file or record does no harm. In other cases, such a maneuver may suspend execution of a program indefinitely while waiting for some process to unlock a file or record. In this case, a conditional lock may be used to print an error code informing the user that the record or file is in use. An ideal strategy might employ both techniques, or use the **alarm** system call to prevent indefinite postponement of file access.

1.15 Locking Schemes

If more than one program is relying on the **lock** system call, a mutually agreed upon scheme must be devised so that all programs use the same identifier to reference records in a file. This identifier is the locking sequence and may contain from one to 16 bytes. An example of a locking sequence is the first 8 bytes of the filename followed by the number of the record to be locked. This scheme works as long as no two files simultaneously in use have names beginning with the same eight characters, and as long as two different processes do not access the same file through two links having different names.

A more elaborate locking scheme uses the file device and inode numbers. The combination of device and inode numbers is a unique file identifier. The number of the device on which a file resides can be obtained by using the {**fstat**} system call. The locking sequence could be composed of a device number followed by an inode number and a record number.

If the number of available locks is exceeded, the operating system returns from a **lock** system call with an error message. This message merely indicates there is no room left in the lock table.

A **_deadlock** error is returned if the operating system detects a deadlock condition.

All locks installed by a process are automatically unlocked when the process is terminated.

1.16 Sample Implementations of Locks

The uses of record locks are best shown through illustration. Consider an inventory management system on a multi-user Cromix system at a music store. If salesperson A sells a guitar and wishes to decrement the inventory record, the program would enter a section of code designed to perform the following functions:

1. Request record number to read.
2. Lock the record with a shared, unconditional lock.
3. Read the record.
4. Unlock the record.

The program might then inform the salesperson that three guitars are in stock. The salesperson rings up the sale, decrements the count of guitars in stock to two, and writes the record to the database using an unshared conditional lock during the write. Difficulties arise if another salesperson, B, also sells a guitar at the same time. B might read the record at the same time as A, decrement the inventory, and write the file out to the database. The record shows that two guitars are in stock, when in fact, there is now only one.

There are several possible solutions to the problem. The simplest is to make an unshared lock at the time of the original read and perform the unlock only after the record had been written out. The problem with this scheme is the potential for barring another user from access to the record for a long time.

A more adequate solution to the problem is to let the system resolve possible conflicts. All user reads are preceded by a shared lock, which permits simultaneous access of the record by other users. When the modified record is to be written out, the system checks to see if the record has been modified in the interim period. If it has not been changed, it is written out. If it has been changed, the value of the record must be recalculated.

1.17 Cromix-Plus Error Numbers

If the Cromix-Plus operating system cannot complete a system call normally, it will return an error. The interface functions in **syslib** are designed so that they always return a particular value, most often the integer (-1). This is used to indicate an error. If an error is returned, the error number is stored in the global integer variable **errno**. Enough information is stored in other global variables to enable the **error** function to write out a decent error message.

1.18 Error Numbers

29	_arglist	The argument list that was provided is too big.
28	_argtable	The argument table is exhausted
69	_badaddress	Illegal address was passed to the system call.
15	_badcall	Illegal system call
1	_badchan	The channel number, that was passed to a system call, was not obtained from the open function.

54	_badformat	A file (typically a **.bin** file) has illegal structure.
42	_badfree	A block is out of range in the free list.
43	_badinum	The inode number is out of range.
52	_badio	An error in doing input or output.
8	_badname	The filename, that was passed to the system call, does not conform to the syntax.
47	_badpipe	An attempt to write to a broken pipe.
34	_badvalue	A value passed to the system call was out of range.
56	_cdossim	The CDOS simulator (**sim.bin**) is required.
40	_chnaccess	An attempt was made to access a channel that was not open for such type of access.
57	_corrupt	System image is corrupted.
49	_deadlock	A possible deadlock condition has been detected.
36	_devopen	A device cannot be open.
31	_difdev	A system call tried to make a link from one device to another.
9	_diraccess	An attempt was made to access a directory and that access was not granted.
37	_diruse	An attempt was made to delete a directory that was not empty.
4	_endfile	End-of-file has been reached.
11	_exists	An attempt has been made to create a file that already exists.
10	_filaccess	An attempt was made to access a file and that access was not granted.
16	_filsize	An attempt was made to create a file too big.
6	_filtable	Too many files open for system, **filcnt** too small.
38	_filuse	A system call requested exclusive access to a file that is currently in use.
22	_fsbusy	A file system cannot be unmounted.
14	_inotable	The inode table is exhausted.
5	_ioerror	A physical data transmission error has occurred.
64	_ipc2big	An IPC facility cannot handle so big an entity.
58	_ipcaccess	The user does have such an access to the IPC facility.
63	_ipcagain	The process would be put to sleep but has asked to return an error instead.
60	_ipcexists	The IPC facility to be created already exists.
61	_ipcnoent	The IPC facility was not found.
65	_ipcnomsg	There is no such message in the IPC message queue.
66	_ipcrange	A value in the IPC system call is out of range.
62	_ipcremove	The IPC facility has been removed.
59	_ipcspace	There is not enough root to create an IPC facility.
19	_isdir	The file referenced is a directory file and the requested operation cannot be done on a directory.
50	_lcktable	There is no room to lock another sequence.
48	_locked	The sequence is already locked and the user asked not to be put to sleep.
17	_mnttable	There is no space to mount another device.

25	_nochild	The child process referenced does not exist.
32	_nodevice	There is no such device.
13	_noinode	There are no free inodes.
39	_nomatch	There is no match on the specified ambiguous pathname.
26	_nomemory	There is not enough free memory to execute the system call.
45	_noproc	The process referenced does not exist.
12	_nospace	There are no free disk blocks.
68	_nostext	There no room to run another shared text program.
21	_notblk	The device referenced is not a block device.
35	_notconn	The requested I/O device is not connected to the system.
41	_notcromix	The block device referenced is incompatible with Cromix-Plus operating system.
18	_notdir	The file referenced is not a directory.
7	_notexist	The file referenced does not exist.
24	_notmount	The device to be unmounted is not mounted.
3	_notopen	The specified channel is not open.
23	_notordin	The specified file is not an ordinary file.
53	_noz80	Z80 programs cannot be run, or the **/etc/z80.bin** simulator was not found.
30	_numlinks	A file can have at most 255 links.
27	_ovflo	Divide system call produced an overflow.
20	_priv	A nonprivileged user made an attempt to execute a privileged operation.
67	_ptable	There is not enough page tables. Increase the **ptbcnt sysdef** parameter.
44	_readonly	The device is mounted for read-only and cannot be written to.
55	_runaway	A runaway Z80 program was aborted.
46	_ssignal	System call was aborted by a signal.
51	_tapeio	There was some kind of tape I/O error.
2	_toomany	The user has too many open files.
33	_usrtable	There are no more process tables available to run another process.

Chapter2 - Cromix-Plus System Call Descriptions

2.1 The Alarm Function

function: alarm
purpose: Send sigalarm to the calling process

user access: all users

include files: <jsysequ.h>
<syslib.h>

synopsis: int alarm(snum)
int snum;

Description

The **alarm** function sends the alarm signal, **sigalarm**, to the current process after **snum** seconds have elapsed. If the call **alarm(0)** is issued after an alarm has been set up, the previous alarm is canceled.

Return value:

0	if no error occurred;
ERR	if an error occurred.

2.2 The Boot Function

function: boot
purpose: Boot new operating system

user access: <jsyseq.h>
<syslib.h>

synopsis: int boot(exadd,size)
unsigned short *exadd; unsigned long size;

Description

This call boots a new operating system. The user program must read the new operating system into his memory. The boot function will shutdown the running system, move **size** bytes from address **exadd** to address 000000, load:

D1.L	size of code
D2.L	current root device

and simulate the reset condition.

Return value:

does not return	if no error occurred;
ERR	if an error occurred,

Common errors:

_priv	The call was issued by a nonprivileged user.
_badaddress	The address passed to the system call does not belong to user's address space.

2.3 The Caccess Function

function:	caccess
purpose:	Test access of a channel.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int caccess(channel,mask) int channel, mask;

Description

Caccess tests the specified open channel for access as specified by mask:

mask	what to check
ac_read	read access
ac_exec	execution access
ac_writ	write access
ac_apnd	append access

More than one value can be "ORed" into mask to check for more than one permission at a time. If the caller has all indicated access permissions, the function returns zero. If the caller lacks some of the indicated access permissions, the value ERR is returned and **errno** indicates the error.

As implemented in the Cromix Operating System, the function does not test the access granted during the open procedure. It tests the access the user could have obtained. In other words, the function works like faccess except that the file is identified by the channel number instead of pathname.

Common errors:

_fileaccess	The caller does not have the access he asked for.
_notopen	The specified channel is not open.

2.4 The Cchstat Function

function: cchstat
purpose: Change status information of an open file.

user access: see below

include files: <jsysequ.h>
<syslib.h>

synopsis: int cchstat(channel,statustype,statusvalue)
int channel, statustype, statusvalue;

or

int cchstat(channel,statustype,statusvalue,statusmask)
int channel, statustype, statusvalue, statusmask;

or

int cchstat(channel,statustype,statustime)
int channel, statustype; struct st_time *statustime;

Description

Cchstat changes various components in the inode which is identified by the channel number. The first two arguments are always the same. The remaining arguments depend on **statustype:**

cchstat(channel,st_owner,statusvalue)

Only a privileged user can change the owner ID of the file to **statusvalue.**

cchstat(channel,st_group,statusvalue)

Only a privileged user can change the group ID of the file to **statusvalue.**

cchstat(channel,st_aowner,statusvalue,statusmask)

Only a privileged user or owner of the file can change the access permissions of the owner. **Statusmask** specifies which bits are to be changed, **statusvalue** specifies new bit values. Both **statusvalue** and **statusmask** should be formed as described below.

cchstat(channel,st_agroup,statusvalue,statusmask)

Only a privileged user or owner of the file can change the

access permissions of the group. **Statusmask** specifies which bits are to be changed, **statusvalue** specifies new bit values. Both **statusvalue** and **statusmask** should be formed as described below.

cchstat(channel,st_aother,statusvalue,statusmask)

Only a privileged user or owner of the file can change the access permissions of the public. **Statusmask** specifies which bits are to be changed, **statusvalue** specifies new bit values. Both **statusvalue** and **statusmask** should be formed as described below.

cchstat(channel,st_stext,statusvalue)

Only a privileged user or owner of the file can change the shared text flag. The low order bit of **statusvalue** is used to define the shared text flag.

cchstat(channel,st_tcreate,statustime)

Only a privileged user can change the time the file was created.

cchstat(channel,st_tmodify,statustime)

Only a privileged user can change the time the file was modified.

cchstat(channel,st_taccess,statustime)

Only a privileged user can change the time the file was accessed.

cchstat(channel,st_tdumped,statustime)

Only a privileged user can change the time the file was dumped.

To change the access permissions **statusmask** and **statusvalue** should be formed from:

ac_read	read permission
ac_exec	execute permission
ac_writ	write permission
ac_apnd	append permission

For example,

statusmask	ac_readlac_writ
statusvalue	ac_read

will change read and write access permission to allow read and disallow write.

The function returns

0	if successful
ERR	if an error occurred

Common errors:

_fileaccess	The caller does not have the access he asked for.
_priv	The user is not a privileged user or he does not own the file.
_notopen	The specified channel is not open.
_badaddress	The address passed to the system call does not belong to user's address space.

2.5 The Ccromix Function

function:	ccromix
purpose:	General system call.
user access:	depends on call
include files:	<jsysequ.h> <syslib.h>
synopsis:	int ccromix(syscall,regs) int syscall; struct sys_reg *regs;

Description

This call implements the general system call. The structure **sys_reg** contains all the registers which take part in any system call. The user should load the **sys_reg** structure with appropriate values and invoke the **ccromix** function to do a system call. The **_error** and **_wrbyte** system calls cannot be used with the **ccromix** function.

The function returns:

0	if successful
ERR	if error

2.6 The Chdup Function

function:	chdup
purpose:	Create a duplicate channel number.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int chdup(channel) int channel;

Description

The **chdup** call duplicates a channel. The function will return the lowest available channel number which can be used instead of the original channel number.

The function returns:

new channel number	if successful
ERR	if error

Common errors:

_notopen	The specified channel is not open.
_toomany	There are no free channels left.

2.7 The Chkdev Function

function:	chkdev
purpose:	Verify presence of a driver.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int chkdev(dtype,majorno,minorno) int dtype, majorno, minorno;

Description

The **chkdev** call verifies the presence of a device driver. The device type should be:

is_block	for block device
is_char	for character device

The function returns:

0	if driver present
ERR	if not

Common errors:

_nodevice	The specified device driver does not exist.

2.8 The Clink Function

function:	clink
purpose:	Establish an additional link to an open file.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int clink(channel,pathname) int channel; char *pathname;

Description

The **clink** call establishes a link from the file open on the specified channel to the new file **pathname**. The new file **pathname** must not exist before the **clink** call is made.

The function returns:

0	if successful;
ERR	if error

Common errors:

_badname	The suggested pathname is an illegal Cromix pathname.
_isdir	A directory cannot be linked.
_numlinks	The file has too many links.
_diraccess	The user does not have the appropriate access to create the new pathname.
_notopen	The specified channel is not open.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.9 The Close Function

function:	close
purpose:	Close an open file.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int close(channel) int channel;

Description

The **close** call flushes all buffers associated with the specified **channel** number and disassociates the **channel** number from the file to which it was assigned. This function is part of **clib.obj.**

The function returns:

0	if successful
ERR	if error

Common errors:

_notopen	The channel to be closed was not open to start with.

2.10 The Create Function

function:	create
purpose:	Create and open a file.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int create(filename,accessmode,accessmask) char *filename; int accessmode, accessmask;

Description

The **create** call creates a new file and opens it for the specified access.

Accessmode defines how the created file will be opened. The user may specify one of the nonexclusive modes

op_read	read only
op_write	write only
op_rdwr	read/write
op_append	append only

or one of the exclusive modes

op_xread	read only
op_xwrite	write only
op_xrdwr	read/write
op_xappend	append only

If a nonexclusive mode is selected the **accessmask** is not used. If an exclusive mode is selected, the bits **op_read**, **op_write**, **op_rdwr**, **op_append** in the **accessmask** are set to prevent such an access from other users.

Example:

To create a file "some" with exclusive read/write permission, use the call

```
create ("some", op_xrdwr, 1<<op_write | 1<<op_rdwr | 1<<op_append)
```

The current process will create the file and open it for read/write. Until the file is closed, other processes may open this file only for read.

Two additional values may be "ORed" into **accessmode** to tell what should happen if the file to be created already exists:

op_truncf	delete existing data

op_condf	return error if file exists

If the file was new, it will have the default access privileges (defined at **crogen** time). The default is rewa.re.re (which means that the owner has all permissions, group and public permissions are for read and execute). The function returns:

channel number of the open file	if successful
ERR	if error

Common errors:

_filtable	Too many files were open in the system.
_badname	Illegal pathname
_diraccess	The user lacks appropriate access to a directory
_badaddress	The address passed to the system call does not belong to the user's address space.

2.11 The Cstat Function

function: cstat
purpose: Return status information of an open file.

user access: all users

include files: <jsysequ.h>
<syslib.h>

synopsis: int cstat(channel,statustype,inodebuffer)
int channel, statustype; char inodebuffer[128];

or

int cstat(channel,statustype)
int channel, statustype;

or

int cstat(channel,statustype,statustime)
int channel, statustype; struct st_time *statustime;

Description

The function **cstat** extracts various components from the inode identified by the channel number. The first two arguments are always the same. The remaining arguments depend on **statustype:**

cstat(channel,st_all,inodebuffer)

Copy all of the inode into 128 bytes **inodebuffer**. Return zero.

cstat(channel,st_owner)

Return the owner ID of the file.

cstat(channel,st_group)

Return the group ID of the file.

cstat(channel,st_aowner)

Return the access mask for the owner.

cstat(channel,st_agroup)

Return the access mask for the group.

cstat(channel,st_aother)

Return the access mask for the public.

cstat(channel,st_stext)

Return the shared text flag.

cstat(channel,st_ftype)

Return the file type (is_ordin, is_direct, is_char, is_block, is_pipe).

cstat(channel,st_size)

Return the file size in bytes.

cstat(channel,st_nlinks)

Return the number of file links.

cstat(channel,st_inum)

Return the inode number.

cstat(channel,st_tcreate,statustime)

Store the time the file was created into the structure pointed to by **statustime**. Return zero.

cstat(channel,st_tmodify,statustime)

Store the time the file was modified into the structure pointed to by **statustime**. Return zero.

cstat(channel,st_taccess,statustime)

Store the time the file was accessed into the structure pointed to by **statustime**. Return zero.

cstat(channel,st_tdumped,statustime)

Store the time the file was dumped into the structure pointed to by **statustime**. Return zero.

cstat(channel,st_devno)

Return the device number of the device specified by channel. If the channel number does not refer to a device file, zero is returned.

cstat(channel,st_pdevno)

Return the device number of the device specified by channel. If the channel number does not refer to a device file zero is returned. If the device number happens to be zero, the device number of the controlling **tty** (character device) or of the root device (block device) will be returned.

cstat(channel,st_device)

Return the device number of the device where the file specified by channel resides.

The access permission returned is build from the values

ac_read	read permission
ac_exec	execute permission
ac_writ	write permission
ac_apnd	append permission

The device numbers returned are built like this:

majorno << 8 | minorno

The function returns

as described above	if successful
ERR	if an error occurred

Common errors:

_notopen	The channel referenced is not open.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.12 The Cxexit Function

function:	cxexit
purpose:	Terminate the current process.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int cxexit(status) int status

Description

Terminate the current process and return process termination status to the parent process. The **wait** function issued by the parent process will return this value as its process termination status.

This function implements the **_exit** system call. The C callable function **exit** as described in the C manual does more than just a simple **cxexit**.

The **cxexit** function does not return.

2.13 The Cxopen Function

function: cxopen
purpose: Open a file.

user access: all users

include files: <jsysequ.h>
<syslib.h>

synopsis: int cxopen(filename,accessmode,accessmask)
char *filename; int accessmode, accessmask;

Description

The **cxopen** call opens the file for the specified access.

Accessmode defines how the file will be opened. The user may specify one of the nonexclusive modes

op_read	read only
op_write	write_only
op_rdwr	read/write
op_append	append only

or one of the exclusive modes

op_xread	read only
op_xwrite	write only
op_xrdwr	read/write
op_xappend	append only

If a nonexclusive mode is selected, the accessmask is not used at all. If an exclusive mode is selected, the bits **op_read**, **op_write**, **op_rdwr**, **op_append** in the accessmask, are set to prevent such access from other users.

Example:

To open the file "some" with exclusive read/write permission, use the call

```
cxopen("some", op_xrdwr, 1<<op_write | 1<<op_rdwr | 1<<op_append)
```

The current process will open the file for read/write. Until the file is closed, other processes may open this file only for read.

The **cxopen** function implements **_open** system call. The **open** function as described in the C manual has different parameters.

The function returns:

channel number of the open file	if successful
ERR	if an error occurred

Common errors:

_filtable	Too many open files.
_badname	The pathname is illegal.
_diraccess	The user lacks proper access to one of the directories.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.14 The Delete Function

function:	delete
purpose:	Delete a directory entry.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int delete(pathname) char *pathname;

Description

The **delete** call attempts to remove the specified directory entry. If the removed directory entry is the last link to the file, the space occupied by the file is released, and the file's contents lost.

The **delete** call requires write permission to the directory from which the directory entry will be removed.

If the file is open at the time the call is made and the specified directory entry is the last link to the file, the directory entry is deleted immediately. The file itself is not deleted until the active process closes the file. In order for a directory to be deleted, it must not

1. Contain any files;
2. Be the current directory for any user;
3. Be the root directory for a device.

The function returns:

0	if successful
ERR	if error

Common errors:

_diraccess	The user lacks proper access to one of the referenced directories.
_notexist	The file to be deleted does not exist.
_badaddress	The address passed to the system call does not belong to user's address space.

2.15 The Error Function

function:	error
purpose:	Display Cromix-Plus error message.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int error(channel) int channel;

Description

Error displays the error message defined by the Cromix system for the value of **errno**, which was loaded the last time a Cromix call returned an error. If the **error** function itself generates an error, the error number will not be saved in **errno.**

The function **error** is part of the **clib.obj** library.

The function returns:

0	if successful
ERR	if error

2.16 The Exchg Function

function:	exchg
purpose:	Exchange contents of two open files.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int exchg(ichannel,ochannel) int ichannel, ochannel;

Description

Exchanges the contents of two open files.

The function returns:

0	if successful
ERR	if error

Common errors:

_notopen	One of the channels was not open.

2.17 The Exec Function

function:	exec
purpose:	Execute a program.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int exec(pathname,argv) char *pathname, *argv[];

Description

The **exec** system call replaces the current code with the code of a new program. If an error is encountered after the original code has been scrapped, the original program quietly terminates.

This implementation of the **exec** system call differs in two aspects from the implementation of the **exec** system call in the older versions of Cromix-Plus (older than 31.11):

- The new code actually overlays the old code so that at no point the old and the new code reside in memory.

- Only the channels **stdin**, **stdout**, and **stderr** are retained as opposed to all channels.

Array **argv** of pointers to the arguments must be terminated by a NULL pointer.

Common errors:

_notexist	The file to be executed does not exist.
_filaccess	The user coes not have execute access to the file to be excuted.
_nomemory	There is not enough memory to load the program.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.18 The Faccess Function

function:	faccess
purpose:	Test access to a file.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int faccess(pathname,mask) char *pathname; int mask;

Description

Faccess tests the specified file for the access as specified by **mask**:

mask	what to check
ac_read	read access
ac_exec	execution access
ac_writ	write access
ac_apnd	append access

More than one value can be "ORed" into **mask** to check for more than one permission at a time. If the caller has all indicated access permissions, the function returns zero. If the caller lacks some of the indicated access permissions, the value ERR is returned and **errno** indicates the error.

Common errors:

_badname	The pathname is not legal.
_filaccess	The user does not have the access he asked for.
_notexist	The file to be tested does not exist.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.19 The Fchstat Function

function: fchstat
purpose: Change status information of a file.

user access: see below

include files: <jsysequ.h>
<syslib.h>

synopsis: int fchstat(pathname,statustype,statusvalue)
char *pathname; int statustype, statusvalue;

or

int fchstat(pathname,statustype,statusvalue,statusmask)
char *pathname; int statustype, statusvalue, statusmask;

or

int fchstat(pathname,statustype,statustime)
char *pathname; int statustype;
struct st_time *statustime;

Description

The **fchstat** system call changes various components in the inode identified by **pathname**. The first two arguments are always the same. The remaining arguments depend on **statustype**:

fchstat(pathname,st_owner,statusvalue)

Only a privileged user can change the owner ID of the file to **statusvalue.**

fchstat(pathname,st_group,statusvalue)

Only a privileged user can change the group ID of the file to **statusvalue.**

fchstat(pathname,st_aowner,statusvalue,statusmask)

Only a privileged user or owner of the file can change the access permissions of the owner. **Statusmask** specifies which bits are to be changed, **statusvalue** specifies new bit values. Both **statusvalue** and **statusmask** should be formed as described below.

fchstat(pathname,st_agroup,statusvalue,statusmask)

Only a privileged user or owner of the file can change the access permissions of the group. **Statusmask** specifies which bits are to be changed. **Statusvalue** specifies new bit values. Both **statusvalue** and **statusmask** should be formed as described below.

fchstat(pathname,st_aother,statusvalue,statusmask)

Only a privileged user or owner of the file can change the access permissions of the public. **Statusmask** specifies which bits are to be changed. **Statusvalue** specifies new bit values. Both **statusvalue** and **statusmask** should be formed as described below.

fchstat(pathname,st_stext,statusvalue)

Only a privileged user or owner of the file can change the shared text flag. The low order bit of **statusvalue** is used to define the shared text flag.

fchstat(pathname,st_protect,statusvalue)

Only a privileged user or owner of the file can change the delete-protect flag. The low order bit of **statusvalue** is used to define the delete-protect flag.

fchstat(pathname,st_tcreate,statustime)

Only a privileged user can change the time the file was created.

fchstat(pathname,st_tmodify,statustime)

Only a privileged user can change the time the file was modified.

fchstat(pathname,st_taccess,statustime)

Only a privileged user can change the time the file was accessed.

fchstat(pathname,st_tdumped,statustime)

Only a privileged user can change the time the file was dumped.

To change the access permissions, **statusmask** and **statusvalue** should be formed from:

ac_read	read permission
ac_exec	execute permission
ac_writ	write permission
ac_apnd	append permission

For example:

statusmask	ac_readlac_writ
statusvalue	ac_read

will change read and write access permission to allow read and disallow write.

The function returns:

0	if successful

ERR	if an error occurred

Common errors:

_filaccess	The user does not have permission to change the file attributes.
_priv	The user must be a privileged user to execute such a call.
_notexist	The file does not exist.
_badname	The file is referenced by an illegal pathname.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.20 The Fexec Function

function:	fexec
purpose:	Fork and execute a program.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int fexec(pathname,argv,sigmask,sigvalues) char *pathname, *argv[]; int sigmask, sigvalues;

Description

The **fexec** call begins execution of a program and returns control to the calling program. The call is similar to the **exec** call, except that a new process is created.

The values of **sigmask** and **sigvalues** define how the child process should respond to signals.

To each signal number there corresponds a bit in the **sigmask** and the **sigvalues**. The mask for signal **sigxxx** is defined as

```
1 << (sigxxx-1)
```

If a bit in **sigmask** is zero, the corresponding bit in **sigvalues** is immaterial. The child process will react to signals in the same way as the parent process:

parent	child
abort	abort
ignore	ignore
trap	abort

If a bit in **sigmask** is nonzero, the child process will react on a signal as defined by the corresponding bit in **sigvalues**:

bit in sigvalues	reaction by the child
0	abort
1	ignore

The child process may issue the signal system call to modify the reaction to the individual signals.

Array **argv** of pointers to the arguments must be terminated by the NULL pointer.

The function returns:

child process id	if successful

ERR	if error

Common errors:

_notexist	The file to be executed does not exist.
_filaccess	The user does not have execute access to the file.
_nomemory	There is not enough memory available to fork this program.
_badname	The program to be forked is referenced by an illegal pathname.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.21 The Flink Function

function:	flink
purpose:	Establish a link to a file.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int flink(oldpath,newpath) char *oldpath, *newpath;

Description

The **flink** call establishes a link from an existing file to the new file pathname. The new file pathname must not exist before the **clink** call is made.

The function returns:

0	operation successful;
ERR	if error

Common errors:

_badname	One of the pathnames is illegal.
_isdir	As a rule, directories cannot be linked.
_numlinks	The file has too many links.
_diraccess	The user needs proper access to the directories involved.
_exists	The pathname to be created already exists.
_notexist	The pathname to be linked does not exist.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.22 The Fshell Function

function:	fshell
purpose:	Fork a Shell process.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int fshell(argv,sigmask,sigvalues) char *argv[]; int sigmask, sigvalues;

Description

The **fshell** call begins execution of a shell and returns control to the calling program. The call is similar to the shell call, except that a new process is created.

The values of **sigmask** and **sigvalues** define how the child process should respond to signals.

To each signal number there corresponds a bit in the **sigmask** and the **sigvalues**. The mask for signal **sigxxx** is defined as

1 << (sigxxx-1)

If a bit in **sigmask** is zero, the corresponding bit in **sigvalues** is immaterial, and the child process will react on signals in the same way as the parent process:

parent	child
abort	abort
ignore	ignore
trap	abort

If a bit in **sigmask** is nonzero, the child process will react to a signal as defined by the corresponding bit in **sigvalues**:

bit in sigvalues	reaction by the child
0	abort
1	ignore

Note that the child process may issue the signal system call to modify the reaction to the individual signals.

In every case **argv[0]** should point to the string "shell" (or "sh").

If you want to execute a program, then

argv[1] --> "-p"
argv[2] --> full program name
argv[3] --> arg[1] of the program
argv[4] --> arg[2] of the program
.....
Last pointer should be zero

If you want to execute a command line, then

argv[1] -->	"-c"
argv[2] -->	command line
argv[3]	0

If you want to execute a command file, then

argv[1] -->	command file name
argv[2]	0

or

argv[1] -->	"-q"
argv[2] -->	command file name
argv[3]	0

In the first form, the commands from the command file will be echoed. In the second form, they will not be echoed.

The function returns:

child process id	if successful
ERR	if error

Common errors:

_nomemory	There is not enough memory to fork another Shell.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.23 The Fstat Function

function: fstat
purpose: Return status information of a file.

user access: all users

include files: <jsysequ.h>
<syslib.h>

synopsis:
```
int fstat(pathname,statustype,inodebuffer)
char *pathname; int statustype; char inodebuffer[128];

or

int fstat(pathname,statustype)
char *pathname; int statustype;

or

int fstat(pathname,statustype,statustime)
char *pathname; int statustype;
struct st_time *statustime;
```

Description

The function **fstat** extracts various components from the inode identified by the pathname. The first two arguments are always the same. The remaining arguments depend on **statustype**:

fstat(pathname,st_all,inodebuffer)

Copy all of the inode into 128 byte **inodebuffer**. Return zero.

fstat(pathname,st_owner)

Return the owner ID of the file.

fstat(pathname,st_group)

Return the group ID of the file.

fstat(pathname,st_aowner)

Return the access mask for the owner.

fstat(pathname,st_agroup)

Return the access mask for the group.

fstat(pathname,st_aother)

Return the access mask for the public.

fstat(pathname,st_stext)

Return the shared text flag.

fstat(pathname,st_protect)

Return the delete-protect flag.

fstat(pathname,st_ftype)

Return the file type (is_ordin, is_direct, is_char, is_block, is_pipe).

fstat(pathname,st_size)

Return the file size in bytes.

fstat(pathname,st_nlinks)

Return the number of file links.

fstat(pathname,st_inum)

Return the inode number.

fstat(pathname,st_tcreate,statustime)

Store the time the file was created into the structure pointed to by **statustime**. Return zero.

fstat(pathname,st_tmodify,statustime)

Store the time the file was modified into the structure pointed to by **statustime**. Return zero.

fstat(pathname,st_taccess,statustime)

Store the time the file was accessed into the structure pointed to by **statustime**. Return zero.

fstat(pathname,st_tdumped,statustime)

Store the time the file was dumped into the structure pointed to by **statustime**. Return zero.

fstat(pathname,st_devno)

Return the device number of the device specified by **pathname**. If the **pathname** number does not refer to a device file zero is returned.

fstat(pathname,st_pdevno)

Return the device number of the device specified by **pathname.** If the **pathname** number does not refer to a device file zero is returned. If the device number happens to be zero the device number of the controlling **tty** (character device) or of the root device (block device) will be returned.

fstat(pathname,st_device)

Return the device number of the device where the file specified by **pathname** resides.

The access permission returned is build from the values

ac_read	read permission
ac_exec	execute permission
ac_writ	write permission
ac_apnd	append permission

The device numbers returned are built like this:

majorno << 8 | minorno

The function returns

as described above	if successful
ERR	if an error occurred

Common errors:

_badname	The file is referenced by an illegal pathname.
_badvalue	Illegal status type.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.24 The Getdate Function

function:	getdate
purpose:	Get current date.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int getdate(date) struct sys_date *date;

Description

The current date as kept by the system is stored in the structure pointed to by **date.**

The function returns:

0	if successful;
ERR	if error.

2.25 The Getdir Function

function:	getdir
purpose:	Return current directory pathname.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int getdir(buffer) char buffer[128];

Description

The pathname to the current directory is stored in **buffer**. The pathname will be terminated by a zero byte.

If the resulting pathname exceeds 128 characters it will be suitably abbreviated.

The function returns:

0	if successful;
ERR	if error.

2.26 The Getgrent Function

function: getgrent, getgrgid, getgrnam, setgrent, endgrent
purpose: read and decode group file.

user access: all users

include files: <grp.h>

synopsis:

```
struct group *getgrent()

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

void setgrent();

void endgrent();
```

Description

Getgrent, **getgrgid**, and **getgrnam** each return a pointer to an object with the group structure (see **/usr/include/grp.h**).

Getgrent when first called, returns a pointer to the first group structure in the file; thereafter it returns a pointer to the next structure in the group file. Successive calls can be used to search the entire file.

The group file is kept open and can be rewound by the **setgrent** function, or closed by the **endgrent** function.

Getgrgid searches from the beginning of the file until a numeric group id matching **gid** is found, and returns the pointer to the particular structure in which it was found.

Getgrnam searches from the beginning of the file until a group name matching **name** is found, and returns the pointer to the particular structure in which it was found.

The functions returning pointers return the NULL pointer if entry is not found.

Note

All data is kept in static memory and each call will overwrite previous data.

2.27 The Getgroup Function

function:	getgroup
purpose:	Return group ID.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int getgroup(idtype) int idtype;

Description

Get the group number of the type **idtype** (**id_effective**, **id_login**, **id_program**).

The function returns:

group number	if successful
ERR	if error

2.28 The Getmode Function

function:	getmode
purpose:	Return characteristics of a device.
user access:	all users
include files:	<jsysequ.h> <syslib.h> <modeequ.h> <bmodeequ.h> <tmodeequ.h>
synopsis:	int getmode(channel,modenumber) int channel, modenumber;

Description

See the **modeequ.h** files for the meaning of mode numbers and mode values.

The function returns:

mode value	if successful
ERR	if error occurred.

2.29 The Getpos Function

function:	getpos
purpose:	Get current file position
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int getpos(channel) int channel;

Description

Get current file position.

The function returns:

file position	if successful
ERR	if error.

Common errors:

_notopen	The channel is not open.

2.30 The Getprior Function

function:	getprior
purpose:	Get process priority.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int getprior()

Description

Get a process priority. The result is in the range -40 .. +40. The value -40 is the highest priority, +40 the lowest.

The function returns:

process priority

2.31 The Getproc Function

function:	getproc
purpose:	Get process ID number.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int getproc()

Description

Getproc is a function which enables a process to obtain its process ID number.

The function returns:

current process ID

2.32 The Getpwent Function

function:	getpwent, getpwuid, getpwnam, setpwent, endpwent
purpose:	read and decode password file.
user access:	all users
include files:	<pwd.h>
synopsis:	struct passwd *getpwent()
	struct passwd *getpwuid(uid) int uid;
	struct passwd *getpwnam(name) char *name;
	void setpwent();
	void endpwent();

Description

Getpwent, **getpwuid**, and **getpwnam** each returns a pointer to an object with the **passwd** structure (see **/usr/include/pwd.h**).

Getpwent when first called returns a pointer to the first **passwd** structure in the file; thereafter it returns a pointer to the next structure in the **passwd** file; so successive calls can be used to search the entire file.

The **passwd** file is kept open and can be rewound by the **setpwent** function, or closed by the **endpwent** function.

Getpwuid searches from the beginning of the file until a numeric user id matching **uid** is found, and returns the pointer to the particular structure in which it was found.

Getpwnam searches from the beginning of the file until a login name matching **name** is found, and returns the pointer to the particular structure in which it was found.

The functions returning pointers return a NULL pointer if the entry is not found.

Note

All data is kept in static memory and each call will overwrite previous data.

2.33 The Gettime Function

function:	gettime
purpose:	Get the current time.
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int gettime(time) struct sys_time *time;

Description

Get current time. The value is stored in the structure pointed to by **time**.

The function returns:

0	if successful
ERR	if erroor

2.34 The Getuser Function

function:	getuser
purpose:	Get the user ID of the calling process
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int getuser(idtype) int idtype;

Description

The value of **idtype** should be one of the following:

id_effective	return effective user id
id_login	return user id from the login file
id_program	return user id of the owner of the program

2.35 The Indirect Function

function:	indirect
purpose:	General system call.
user access:	depends on call
include files:	<jsysequ.h> <syslib.h>
synopsis:	int indirect(syscall,regs) int syscall; struct sys_reg *regs;

Description

This call implements the **general system call**. The structure **sys_reg** contains all of the registers which take part in any system call. The user should load the **sys_reg** structure with appropriate values and invoke the **indirect** function to do a system call. See description of assembler system calls for details. The **_wrbyte** and **_error** system calls cannot be used with the **indirect** function.

The function returns:

0	if successful
ERR	if error

2.36 The Kill Function

function:	kill
purpose:	Send the specified signal to the specified process
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int kill(pid,stype) int pid, stype;

Description

The **kill** function sends a signal to a process.

When any signal is received by a process, the process is aborted unless the signal system call specifies that a subroutine be executed, or that the signal be ignored.

When a signal is received, unless it is ignored, an unsatisfied request for input or output from a character device is cancelled. Examples: reading a buffered line from a console or writing a line to the printer.

If a signal is sent to process 0, the same type of signal is sent to all processes that belong to the user invoking the call.

If the user is a privileged user and a **siguser** signal is sent to process 1, system shutdown is initiated.

If a **sigabort** signal is sent to process 1, the **/etc/ttys** file is reexamined. If an entry has a 0 in the leftmost column, the appropriate terminal is logged off and all of its processes are terminated. If an entry shows a 1 in that column, the terminal is logged in if it is not already logged in.

The function returns:

0	if successful
ERR	if error

Common errors:

_priv	Only a privileged user can send signals to processes he did not initiate.
_noproc	Such a process does not exist.

2.37 The Lock Function

function:	lock
purpose:	Lock out the process
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int lock(lock_sequence,ltype,llength) char *lock_sequence; int ltype, llength;

Description

Lock makes (or attempts to make) a lock table entry. If (ltype & 1) is "true", the lock is shared. If (ltype & 2) is "true", the lock is conditional.

The lock will "fail" if the following is true:

> The **lock_sequence** has already been locked and either this lock is meant to be unshared or the sequence is aready locked as unshared.

In the opposite case the lock will "succeed".

If the lock "fails" and the lock is conditional, **errno** is set to **_locked** and the value ERR is returned immediately.

If the lock "fails" and the lock is unconditional, the process is put to sleep until the lock can "succeed".

The function returns:

0	operation successful;
ERR	if error.

Common errors:

_locked	The sequence is already locked.
_deadlock	Locking the sequence would result in a deadlocked situation.
_lcktable	There are no more lock table entries available.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.38 The Makdev Function

function:	makdev
purpose:	Create a device file
user access:	privileged user
include files:	<jsysequ.h> <syslib.h>
synopsis:	int makdev(pathname,dtype,majorno,minorno) char *pathname; int dtype, majorno, minorno;

Description

This call creates a device file. The value returned is:

0	if successful
ERR	if error

Common errors:

_badname	Pathname points to illegal path name.
_exists	The file identified by the pathname already exists.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.39 The Makdir Function

function:	makdir
purpose:	Create a directory file
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int makdir(pathname) char *pathname;

Description

Makes a specified directory. The function returns:

0	if successful
ERR	if error

Common errors:

_badname	Pathname points to an illegal path name.
_exists	The file identified by pathname already exists.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.40 The Memory Function

function:	memory
purpose:	Allocate memory to the process
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int memory(mode,paddr,size,mask) int mode; unsigned char **paddr; unsigned long size, mask;

Description

The memory system call will allocate or deallocate user memory:

mode	action
0	Allocate size bytes according to **mask** and store the pointer to allocated memory into ***paddr.**
1	Deallocate size bytes of user memory pointed to by ***paddr.**

The mask value used for allocating can be used to get memory aligned according to **mask:** the resulting pointer, if masked with **mask**, will be zero. The normal value of **mask** should be zero (no special requirements). If, for example, the mask of 0xffff is used, the allocated memory will be at a 64K boundary.

Only the memory obtained from the **memory** call, mode 0, can be deallocated by the **memory** call, mode 1.

All memory obtained from one call will be contiguous. Two consecutive calls of **memory** will not necessarily return contiguous pieces of memory.

The function will return:

0	if successful;
ERR	if error.

Common errors:

_nomemory	There is not enough memory to fulfill the request.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.41 The Mount Function

function:	mount
purpose:	Enable access to a file system
user access:	privileged user
include files:	<jsysequ.h> <syslib.h>
synopsis:	int mount(dummypath,devpath,access) char *dummypath, *devpath; int access;

Description

Mount a file system on the device identified by **devpath. Dummypath** should be the pathname of an arbitrary file. After a successful mount, the **dummypath** will be the directory identifying the root of the mounted device.

Access should be:

0	read/write
1	read only

The function returns:

0	if succesful;
ERR	if error.

Common errors:

_mnttable	Too many mounted devices.
_fsbusy	The device to be mounted is currently in use.
_notblk	The device to be mounted is not a block device.
_badname	One of the pathnames is illegal.
_notexist	One of the files quoted does not exist.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.42 The Msgctl Function

function:	msgctl
purpose:	Message queue control operations
user access:	all users
include files:	<jsysequ.h> <syslib.h> <ipc.h> <msg.h>
synopsis:	int msgctl(msqid,cmd,buf) int msqid, cmd; struct msqid_ds *buf;

Description

Msgctl provides a variety of message queue control operations as specified by **cmd**. The following commands are available:

IPC_STAT — Place the current value of each member of the data structure associated with the message queue identifier **msqid** into the structure pointed to by **buf**.

IPC_SET — Set the values of the following members of the data structure associated with **msqid** to the corresponding values found in the **msqid_ds** structure pointed to by **buf**:

msg_perm.uid
msg_perm.gid
msg_perm.mode (low order 9 bits only)
msg_qbytes

This **cmd** can only be executed by the super user or by a process that has an effective user ID equal to the **msg_perm.uid** in the data structure associated with the **msqid**. Only the super user can raise the value of **msg_qbytes**.

IPC_RMID — Remove the message queue identifier specified by **msqid** from the system and destroy the message queue and data structure associated with it. This command can only be executed by a privileged user or by the creator of the message queue.

The function returns:

0	if succesful;
ERR	if error.

Common errors:

_badvalue	Invalid command or **msqid** not a valid message queue identifier.
_ipcaccess	Operation permission is denied to the calling process.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.43 The Msgget Function

function:	msgget
purpose:	Get a message queue identifier
user access:	all users
include files:	<jsysequ.h> <syslib.h> <ipc.h> <msg.h>
synopsis:	int msgget(key,msgflg) long key; int msgflg;

Description

Msgget returns the message queue identifier associated with **key**.

A message queue identifier and associated message queue and data structure, are created for **key** if one of the following are true:

> **Key** is equal to IPC_PRIVATE.
>
> **Key** does not already have a message queue identifier associated with it, and (msgflg & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

> **Msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, **msg_perm.gid** are set to be equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low order nine bits of **msg_perm.mode** are set equal to the low order nine bits of **msgflg**.
>
> **Msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, **msg_rtime** are set equal to zero.
>
> **Msg_ctime** is set to be equal to the current time.
>
> **Msg_qbytes** is set to be equal to the system limit.

The function returns:

a non-negative message queue identifier	if successful;
ERR	if error.

Common errors:

_ipcaccess	A message queue identifier exists for the key, but the operation permission as specified by the low order nine bits of **msgflg** will not be granted.
_ipcnoent	A message queue identifier does not exist for key and the create bit in **msgflg** is not set.
_ipcspace	There is no space in the system to create another message queue identifier.
_ipcexists	A message queue identifier exists for the key but both the create and the exclusive bits in the **msgflg** are set.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.44 The Msgrcv Function

function:	msgrcv
purpose:	Receive a message from a message queue
user access:	all users
include files:	<jsysequ.h> <syslib.h> <ipc.h> <msg.h>
synopsis:	int msgrcv(msqid,msgp,msgsz,msgtyp,msgflg) int msqid, msgsz, msgflg; long msgtyp; struct msgbuf *msgp;

Description

Msgrcv reads a message from the message queue associated with the message queue identifier **msqid** and places it in the structure pointed to by **msgp**. The structure is composed of the following members:

```
long    mtype;    /* Message type    */
char    mtext[];  /* Message text    */
```

Mtype is the received message's type as specified by the sending process. **Mtext** is the text of the message. **Msgsz** specifies the size in bytes of **mtext**. The received message is truncated to **msgsz** bytes if it is larger than msgsz and (msgflg & MSG_NOERROR) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

Msgtyp specifies the type of message requested as follows:

If **msgtyp** is equal to zero, the first message in the queue is received.

If **msgtyp** is greater than zero, the first message of the type **msgtyp** is received.

If **msgtyp** is less than zero, the first message of the lowest type that is less than or equal to the absolute value of **msgtyp** is received.

Msgflg specifies the action to be taken if a message of the specified type is not on the queue. These are as follows:

If (msgflg & IPC_NOWAIT) is "true", the calling process will return immediately with the return value of ERR and errno set to **_ipcnomsg**.

If (msgflg & IPC_NOWAIT) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

Msqid is removed from the system. When this occurs, **errno** is set to **_ipcremove**, and value ERR is returned.

The calling process receives a signal that is to be caught. In this case a message is not received, **errno** is set to **_ssignal**, and the value ERR is returned.

Upon successful completion the following actions are taken with respect to the data structure associated with **msqid**:

Msg_qnum is decremented by one.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime is set equal to the current time.

The function will return:

The number of bytes actually placed into mtext	if successful
ERR	if error.

Common errors:

_badvalue	**Msqid** is not a valid message queue identifier.
_ipcaccess	Operation permission is denied to the calling process.
_badvalue	**Mtype** is less than one.
_ipc2big	Message to be received is longer than **msgsz** and the truncate bit in **msgflg** is not set.
_ipcagain	No message of the required type is waiting right now and the nowait bit is set in the **msgflg**.
_badvalue	Message size is less than zero.
_ipcremove	While the system call was waiting to receive the message, the message queue was removed from the system.
_ssignal	A signal was received by the process while it was waiting for the message to be received.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.45 The Msgsnd Function

function: msgsnd
purpose: Send a message to a message queue

user access: all users

include files: <jsysequ.h>
<syslib.h>
<ipc.h>
<msg.h>

synopsis: int msgsnd(msqid,msgp,msgsz,msgflg)
int msqid, msgsz, msgflg; struct msgbuf *msgp;

Description

Msgsnd is used to send a message to the message queue associated with the message queue identifier **msqid**. **Msgp** points to a structure containing the message. The structure is composed of the following members:

```
long    mtype;      /* Message type    */
char    mtext[];    /* Message text    */
```

Mtype is a positive integer that can be used by the receiving process for message selection (see **msgrcv**). **Mtext** is any text of length **msgsz** bytes. **Msgsz** can range from zero to a system imposed maximum.

Msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes**.

The total number of messages on all queues system wide is equal to the system imposed limit.

These actions are as follows:

If (**msgflg** & IPC_NOWAIT) is "true", the calling process will return immediately with the return value of ERR and errno set to **_ipcspace**.

If (msgflg & IPC_NOWAIT) is "false", the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system. When this occurs, **errno** is set to **_ipcremove**, and value ERR is returned.

The calling process receives a signal that is to be caught. In this case a message is not sent, **errno** is set to **_ssignal**, and the value ERR is returned.

Upon successful completion, the following actions are taken with respect to the data structure associated with **msqid**:

Msg_qnum is incremented by one.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime is set equal to the current time.

The function will return:

0	if successful;
ERR	if error.

Common errors:

_badvalue	**Msqid** is not a valid message queue identifier.
_ipcaccess	Operation permission is denied to the calling process.
_badvalue	**Mtype** is less than one.
_ipcagain	Message cannot be sent right now and the nowait bit is set in the **msgflg**.
_badvalue	Message size is less than zero or grater than the system imposed limit.
_ipcremove	While the system call was waiting to get the resources the message queue was removed from the system.
_ssignal	A signal was received by the process while it was waiting for resources to send the message.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.46 The Pause Function

function: pause
purpose: Wait for any signal

user access: all users

include files: <jsysequ.h>
<syslib.h>

synopsis: int pause()

Description

Pause suspends execution of the process until a signal arrives.

Returns:

ERR and **errno** set to **_ssignal**

2.47 The Phys Function

function:	phys
purpose:	Change user access to system memory
user access:	privileged user
include files:	<jsysequ.h> <syslib.h>
synopsis:	int phys(addr,size,access) char *addr; int size, access;

Description

This function changes user access privileges to system memory. **Addr** is the starting address where the access is to be changed. It must be a multiple of page size (4096). **Size** is the size of address range that will have the access changed. **Size** must be again a multiple of page size (4096). **Access** is the the combination of access privileges:

0x02	read access
0x04	write access
0x08	execute access

The function must be used with great caution. If used it will allow the user arbitrary access anywhere in memory. The function is primarily intended to allow specialized user programs the access to such areas as the I/O space and graphics memory.

The function returns int:

0	if successful
ERR	if error

Common errors:

_badvalue	Unreasonable value passed as an argument.
_priv	You must be a privileged user to use this call.

Example

To use the Baseline package with 1/2 Megabyte of graphic memory at address 0x800000 you have to use the code.

```
if (phys(0x800000,0x080000,6) ||
    phys(0xfff000,0x001000,6))
        error(STDERR), exit(ERR);
```

The first line gives you read and write access to graphic memory,
the second line gives you read and write access to the I/O space.

2.48 The Pipe Function

function:	pipe
purpose:	Create a pipe
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int pipe(pipeout,pipein) int *pipeout, *pipein;

Description

The function **pipe** creates a pipe. If there is no error, the function returns zero and two channel numbers in **pipein** and **pipeout**. If there is an error, the function returns ERR.

Note: **Pipeout** should be used for writing, **pipein** should be used for reading.

Common errors:

_toomany	Too many open channels.

2.49 The Popen Function

function:	popen, pclose
purpose:	Initiate pipe to/from a process.
user access:	all users
include files:	<stdio.h>
synopsis:	FILE *popen(command,type) char *command, *type; int pclose(stream) FILE *stream;

Description

The arguments to **popen** are pointers to null-terminated strings containing, respectively, a Shell command line and an I/O mode, either "r" for reading or "w" for writing. **Popen** creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is "w", by writing to the file stream; and one can read from the standard output of the command, if the I/O mode is "r", by reading from the file stream.

A stream opened by **popen** should be closed by **pclose** which waits for the associated process to terminate and returns the exit status of the command.

Popen will return NULL pointer if the files or processes cannot be created, or if the Shell cannot be accessed.

Pclose returns **-1** if the stream is not associated with a "popened" command.

2.50 The Ptrace Function

function: ptrace
purpose: Trace execution of another process.

user access: all users

include files: <jsysequ.h>
<syslib.h>
<ptrace.h>

synopsis: int ptrace(command,pid,addr,data,cnt)
int command, pid, cnt; unsigned char *addr, *data;

Description

Ptrace system call is intended to be used in debuggers like **Ddt**. The system call has a number of subfunctions selected by the first argument (command):

P_START — The parent process (debugger) issues this call to notify the system that the next **fexec** (fshell) system call will fork a debugged process. The debugged process does not start execution by itself. It is waiting in a suspended state until the parent process issues a P_RUN, or a P_TRACE, or a P_TERM **ptrace** function. (The debugged process behaves as if it just hit a breakpoint).

The **pid**, **addr**, **data**, **cnt** arguments are not used.

P_RDSEQ — When the debugged process is in the suspended state, this call will read **cnt** bytes from the (absolute) address **data**, belonging to process **pid**, into caller's memory at **addr**. The specified process must be started with the P_START function preceding the **fexec** call.

P_WRSEQ — When the debugged process is in the suspended state, this call will write **cnt** bytes from the caller's memory address to the (absolute) address **data**, belonging to the process **pid**. The specified process must be started with the P_START function preceding the **fexec** call.

P_RDSTA — When the debugged process is in the suspended state, this call will read all of the process **pid** registers (see **ptrace.h**) into the parent address **addr**. The specified process must be started with the P_START function preceding the **fexec** call.

The **data** and **cnt** arguments are not used.

P_WRSTA — When the debugged process is in the suspended state, this call

will write all of the process **pid** registers (see **ptrace.h**) from the parent address **addr**. The specified process must be started with the P_START function preceeding the **fexec** call.

The **data** and **cnt** arguments are not used.

P_RUN — When the debugged process is in the suspended state, this call will restart the process **pid**. The parent process will normally install breakpoints before issuing this call. Breakpoint can be installed by patching the child code with the TRAP 5 instruction. If the child process executes the TRAP 5 instruction it will go into a suspended state. The system will notify the parent process by sending him the **sigtrace** signal. The specified process must be started with the P_START function preceding the **fexec** call.

The **addr**, **data**, and **cnt** arguments are not used.

P_TRACE — When the debugged process is in the suspended state, this call restarts the process **pid** for the duration of one instruction. After one instruction has been executed the system will notify the parent process by sending it the **sigtrace** signal. The specified process must be started with the P_START function preceding the **fexec** call.

The **addr**, **data**, and **cnt** arguments are not used.

P_TERM — When the debugged process is in the suspended state, this call terminates the process **pid**. The specified process must be started with the P_START function preceding the **fexec** call.

The **addr**, **data**, and **cnt** arguments are not used.

The function returns

0	if no error;
ERR	if error.

Common errors:

_badvalue	Bad command argument.
_badaddress	Bad address value.
_noproc	There is no such process.

2.51 The Rand48 Function

function: drand48, erand48, irand48, krand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48

purpose: generate uniformly distributed pseudo-random numbers

user access: all users

synopsis:

```
double drand48()

double erand48(s)
unsigned short s[3];

long irand48(m)
unsigned short m;

long krand48(s,m)
unsigned short s[3], m;

long lrand48()

long nrand48(s)
unsigned short s[3];

long mrand48()

long jrand48(s)
unsigned short s[3];

void srand48(seedval)
long seedval;

unsigned short *seed48(s)
unsigned short s[3];

void lcong48(param)
unsigned short param[7];
```

Description

This family of functions generates pseudo-random numbers using well known linear congruential algorithm and 48-bit integer arithmetic.

Functions **drand48** and **erand48** return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions **irand48** and **krand48** return non-negative long integers uniformly distributed over the interval [0, m-1].

Functions **lrand48** and **nrand48** return non-negative long integers uniformly distributed over the interval [0, 2 ** 31).

Functions **mrand48** and **jrand48** return signed long integers uniformly distributed over the interval [-2 ** 31, 2 ** 31).

Functions **srand48**, **seed48** and **lcong48** are initialization entry points, one of which should be invoked before either **drand48**, **irand48**, **lrand48** or **mrand48** is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if **drand48**, **irand48**, **lrand48** or **mrand48** is called without a prior call to an initialization entry point.) Functions **erand48**, **krand48**, **nrand48** and **jrand48** do not require an initialization entry point to be called first.

All the routine work by generating a sequence of 48-bit integer values, X[i], according to the linear congruential formula

$$X[n+1] = (a * X[n] + c) \% M, \qquad n >= 0$$

The parameter M = 2 ** 48; hence 48 bit integer arithmetic is performed. Unless **lcong48** has been invoked, the multiplier value **a** and the addend value **c** are given by:

a = 0x5DEECE66D
c = 0xB

The value returned by any of the functions **drand48**, **erand48**, **irand48**, **krand48**, **lrand48**, **nrand48**, **mrand48** or **jrand48** is computed by first generating the next 48-bit X[i] in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high order (left-most) bits of X[i] and transformed into the returned value.

The functions **drand48**, **irand48**, **lrand48** and **mrand48** store the last 48-bit X[i] generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions **erand48**, **krand48**, **nrand48** and **jrand48** require the calling program to provide storage for successive X[i] values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the initial value of the X[i] into the array and pass it as an argument. By using different arguments, functions **erand47**, **krand48**, **nrand48** and **jrand48** allow separate modules of a large program to generate several independent streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will not depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function **srand48** sets the high-order 32 bits of X[i] to the 32 bits contained in its argument. The low order 16 bits are set to the arbitrary value 0x330E.

The initializer function **seed48** sets the value of X[i] to the 48-bit value specified in the argument array. In addition, the previous value of X[i] is copied into a 48-bit internal buffer, used only by **seed48**, and a pointer to this buffer is the value returned by **seed48**. This returned pointer, which can be just ignored if not needed, is useful if a program is to be restarted from a given point at some future time - use the pointer to get at and store the last X[i] value, and then use this value to reinitialize via **seed48** when the program is restarted.

The initialization function **lcong48** allows the user to specify the initial X[i], the multiplier value **a**, and

the added value **c**. Argument array elements param[0..2] specify X[i], param[3..5] specify the multiplier **a**, and param[6] specifies the 16-bit addend **c**. After **lcong48** has been called, a subsequent call to either **srand48** or **seed48** will restore the "standard" multiplier and addend values, **a** and **c**, as specified earlier.

2.52 The Rdbyte Function

function:	rdbyte
purpose:	Read a byte
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int rdbyte(channel) int channel;

Description

Read next byte from channel.

Returns:

byte read	if successful
ERR	if error

Common errors:

_notopen	The channel to read from is not open.
_filaccess	The user does not have read access to the file to read from.
_ioerror	Any kind of driver error, diagnosed on the raw terminal.
_endfile	The file is positioned at the end of the file.
_ssignal	A signal was received while waiting for a byte.

2.53 The Rdline Function

function:	rdline
purpose:	Send a line
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int rdline(channel,buffer,maxbytes) int channel, maxbytes; char *buffer;

Description

Read bytes until either:

'\n' is read
'\0' is read
maxbytes are read
a signal is received

Returns:

number of bytes read	if successful
ERR	if error

Common errors:

_notopen	The channel to read from is not open.
_filaccess	The user does not have read access to the file to read from.
_ioerror	Any kind of driver error, diagnosed on the raw terminal.
_endfile	The file is positioned at the end of the file.
_ssignal	A signal was received while waiting for a byte.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.54 The Rdseq Function

function:	rdseq
purpose:	Read sequential bytes
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int rdseq(channel,buffer,bytecount) iint channel, bytecount; char *buffer;

Description

Reads bytes until:

bytecount bytes are read
a signal is received

Returns:

number of bytes read	if successful
ERR	if error

Common errors:

_notopen	The channel to read from is not open.
_filaccess	The user does not have read access to the file to read from.
_ioerror	Any kind of driver error, diagnosed on the raw terminal.
_endfile	The file is positioned at the end of the file.
_ssignal	A signal was received while waiting for a byte.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.55 The Semctl Function

function: semctl
purpose: Semaphore control operations

user access: all users

include files:
```
<jsysequ.h>
<syslib.h>
<ipc.h>
<sem.h>
```

synopsis:
```
int semctl(semid,semnum,cmd,arg)
int semid, semnum, cmd;
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short array[];
} arg;
```

Description

Semctl provides a variety of semaphore control operations as specified by **cmd**.

The following commands are executed with respect to the semaphore specified by **semid** and **semnum**:

GETVAL	Return the value of **semval**.
SETVAL	Set the value of **semval** to **arg.val**. When this command is succesfully executed the **semadj** value corresponding to the specified semaphore in all processes is cleared.
GETPID	Return the value of **sempid**.
GETNCNT	Return the value of **semncnt**.
GETZCNT	Return the value of **semzcnt**.

The following **cmds** return and set, respectively, every **semval** in the set of semaphores.

GETALL	Place **semvals** into array pointed to by **arg.array**.
SETALL	Set **semvals** according to the array pointed to by **arg.array**. When this command is succesfully executed the **semadj** values corresponding to each specified semaphore in all processes are cleared.

The following **cmds** are also available:

IPC_STAT — Place the current value of each member of the data structure associated with **semid** into the structure pointed to by **arg.buf**.

IPC_SET — Set the values of the following members of the data structure associated with **semid** to the corresponding values found in the **semid_ds** structure pointed to by **arg.buf**:

sem_perm.uid
sem_perm.gid
sem_perm.mode (low order 9 bits only)

This **cmd** can only be executed by the super user or by a process that has an effective user ID equal to the **sem_perm.uid** in the data structure associated with the **semid**.

IPC_RMID — Remove the semaphore identifier specified by **semid** from the system and destroy the set of semaphores and data structure associated with it. This command can only be executed by a super user or by the creator of the semaphore group.

Upon successful completion, the value returned depends on the **cmd** as follows:

GETVAL	The value of **semval**.
GETPID	The value of **sempid**.
GETNCNT	The value of **semncnt**.
GETZCNT	The value of **semzcnt**.
All others	A value of zero.

Otherwise, a value ERR is returned and **errno** is set to indicate the error.

Common errors:

_badvalue	**Semid** is not a valid shared memory identifier.
_ipcaccess	Operation permission is denied to the calling process.
_badvalue	Invalid command.
_ipcrange	An invalid semaphore number is used.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.56 The Semget Function

function:	semget
purpose:	Get a semaphore identifier
user access:	all users
include files:	<jsysequ.h> <syslib.h> <ipc.h> <sem.h>
synopsis:	int semget(key,nsems,semflg) long key; int nsems, semflg;

Description

Semget returns the semaphore identifier associated with **key**.

A semaphore identifier and associated data structure and set containing **nsems** semaphores are created for **key** if one of the following are true:

> **Key** is equal to IPC_PRIVATE.
>
> **Key** does not alredy have a semaphore identifier associated with it, and (**semflg** & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

> **Sem_perm.cuidf**, **sem_perm.uid**, **sem_perm.cgid**, **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low order nine bits of **sem_perm.mode** are set equal to the low order nine bits of **semflg**.
>
> **Sem_nsems** is set equal to the value of **nsems**.
>
> **Sem_otime** is set to zero and **sem_ctime** is set equal to the current time.

The function returns:

a nonnegative semaphore identifier	if successful;
ERR	if error.

Common errors:

_ipcaccess	A semaphore group identifier exists for the key but

	operation permission as specified by the low order nine bits will not be granted.
_ipcnoent	A semaphore group identifier does not exist for key and the create bit in flags is not set.
_ipcspace	There is no space in the system to create another semaphore group identifier.
_ipcexists	A semaphore group identifier exists for the key but both create and exclusive bits in flags are set.
_badvalue	The number of semaphores specified is either not positive, or greater than system imposed limit, or greater then the existing number in case the semaphore group identifier alredy exists.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.57 The Semop Function

function: semop
purpose: Execute semaphore operations

user access: all users

include files: <jsysequ.h>
<syslib.h>
<ipc.h>
<sem.h>

synopsis: int semop(semid,sops,nsops)
int semid, nsops; struct sembuf sops;

Description

Semop is used to automatically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier **semid**. **Sops** is a pointer to the array of semaphore-operation structures. **Nsops** is the number of such structures in the array. The contents of each structure includes the following members:

```
unsigned short  sem_num;  /* Semaphore number    */
short           sem_op;   /* Semaphore operation */
short           sem_flg;  /* Operation flags     */
```

Each semaphore operation specified by **sem_op** is performed on the corresponding semaphore specified by **semid** and **sem_num**.

Sem_op specifies one of the three semaphore operations as follows:

If **sem_op** is a negative integer, one of the following will occur:

If **semval** is greater than or equal to the absolute value of **sem_op**, the absolute value of **sem_op** is subtracted from **semval**. Also, if (sem_flg & SEM_UNDO) is "true", the absolute value of **sem_op** is added to the calling process's **semadj** value for the specified semaphore.

If **semval** is less than the absolute value of **sem_op** and (sem_flg & IPC_NOWAIT) is "true", **semop** will return immediately.

If **semval** is less than the absolute value of **sem_op** and (sem_flg & IPC_NOWAIT) is "false", **semop** will increment the **semncnt** associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

Semval becomes greater than or equal to the absolute value of **sem_op**. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, the absolute value of **sem_op** is subtracted from **sem_val** and, if (sem_flg & SEM_UNDO) is "true", the absolute value of **sem_op** is added to the calling process's **semadj** value for the specified semaphore.

The **semid** for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set to **_ipcremove**, and a value of ERR is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, and the calling process resumes execution. **Errno** is set to **_ssignal** and a value of ERR is returned.

If **sem_op** is a positive integer, the value of **sem_op** is added to **semval** and, if (sem_flg & SEM_UNDO) is "true", the value of **sem_op** is subtracted from the calling process's **semadj** value for the specified semaphore.

If **sem_op** is zero, one of the following will occur:

If **semval** is zero, **semop** will return immediately.

If **semval** is not equal to zero and (sem_flg & IPC_NOWAIT) is "true", **semop** will return immediately.

If **semval** is not equal to zero and (sem_flg & IPC_NOWAIT) is "false", **semop** will increment the **semzcnt** associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of **semzcnt** associated with the specified semaphore is decremented.

The **semid** for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set to **_ipcremove**, and a value of ERR is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of **semzcnt** associated with the specified semaphore is decremented, and the calling process resumes execution. **Errno** is set to **_ssignal** and a value of ERR is returned.

Upon successful completion, the value of **sempid** for each semaphore specified in the array **sops** is set equal to the process ID of the calling process.

The function returns:

0	if successful;
ERR	if error.

Common errors:

_badvalue	**Semid** is not a valid semaphore group identifier.
_ipcaccess	Operation permission is denied to the calling process.
_ipcspace	The system imposed limit on the number of undo structures that a process can use, has been reached.
_ipcagain	The process would be put to sleep, but (sem_flg & IPC_NOWAIT) is nonzero.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.58 The Setdate Function

function:	setdate
purpose:	Set current date
user access:	privileged user
include files:	<jsysequ.h> <syslib.h>
synopsis:	int setdate(date) struct sys_date *date;

Description

This function (must be issued by a privileged user) sets the system date. The day of the week need not be specified.

The function returns:

0	if successful
ERR	if error

Common errors:

_priv	Only a privileged user can set the system date.

2.59 The Setdir Function

function:	setdir
purpose:	Change current directory
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int setdir(pathname) char *pathname;

Description

Change current directory to **pathname**. Returns:

0	if successful
ERR	if error

Common errors:

_notdir	The path name specified does not identify a directory.
_diraccess	The user has no execute access into the directory.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.60 The Setgroup Function

function: setgroup
purpose: Change group ID

user access: all users

include files: <jsysequ.h>
<syslib.h>

synopsis: int setgroup(idtype,idvalue,idnumber)
int idtype, idvalue, idnumber;

Description

This call sets the chosen group id (**id_effective**, **id_program**, **id_login**) to the value specified by **idvalue**:

idvalue	new id
id_effective	present effective id
id_login user	login id
id_program	program owner id
id_number	idnumber specified

The function returns:

0	if successful
ERR	if error

Common errors:

_priv	Only a privileged user can set the group to an arbitrary number.

2.61 The Setjmp and Longjmp Functions

function:	setjmp, longjmp
purpose:	Provides returns from somewhere deep in the C program
user access:	all users
include files:	<setjmp.h>
synopsis:	int setjmp(env) jmp_buf env; longjmp(env,val) jmp_buf env; int val;

Description

Functions **setjmp** and **longjmp** can be used to organize a premature return from somewhere deep in the sequence of C functions.

The function **setjmp** saves its stack environment into **env** for later use by the function **longjmp**. The **setjmp** function returns the value zero.

The function **longjmp** restores the environment saved into **env** by a call to the function **setjmp**. It then returns in such a way that execution continues as if the call of **setjmp** had just returned the nonzero value **val** to the function that invoked **setjmp**. It must not have itself returned in the interim. All accessible data have the values as of the time **longjmp** was called.

2.62 The Setlev Function

function:	setlev
purpose:	Set interrupt level of the processor
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int setlev(level) int level;

Description

The **setlev** function sets the interrupt level of the process to prescribed level thereby disabling interrupts of a given or lower level to occur. Interrupt level cannot be set to a value that exceeds the **sysdef** parameter **Maxlev**. System administrators are strongly encouraged to set **Maxlev** to zero thereby effectively disabling the **setlev** function.

If the interrupt level is set to a nonzero value strange things may occur:

- the process will become not abortable
- all other processes may be suspended

The changed interrupt level will stay in effect (for this process) until reset to zero. If the user program that set the interrupt level to a nonzero value executes any system call, the system will take over the interrupt handling for the duration of the system call. This means that during the system call interrupts will be enabled and even process switching may occur. When the user process regains control the interrupt level will be set back to the user value.

We would like to point out that users that do need to use the **setlev** function should write their own driver to do the job.

Return value:

0	if no error occurred;
ERR	if an error occurred.

2.63 The Setmode Function

function:	setmode
purpose:	Change characteristics of a device
user access:	all users
include files:	<jsysequ.h> <syslib.h> <modeequ.h> <bmodeequ.h> <tmodeequ.h>
synopsis:	int setmode(channel,modenumber,modevalue,modemask) int channel, modenumber, modevalue, modemask;

Description

See **modeequ.h** files for the mode number and for the meaning of mode values.

The function returns:

old mode value	if successful
ERR	if error

Common errors:

_badvalue	Invalid mode number.

2.64 The Setpos Function

function: setpos
purpose: Change file position

user access: all users

include files: <jsysequ.h>
<syslib.h>

synopsis: int setpos(channel,filepointer,mode)
int channel, filepointer, mode;

Description

The current file position is set according to **mode**:

fwd_begin	**filepointer** bytes from the beginning
fwd_current	**filepointer** bytes from the current position
fwd_end	**filepointer** bytes from the end of a file
bak_currnet	**filepointer** bytes back from the beginning
bak_end	**filepointer** bytes back from the end of a file

Note: **Filepointer** must be nonnegative.

Returns:

0	if successful
ERR	if error

Common errors:

_notopen	The channel is not open.
_notblk	The channel does not reference a file or a block device.

2.65 The Setprior Function

function:	setprior
purpose:	Set process priority
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int setprior(priority) int priority;

Description

Set process priority to **priority**. Only a privileged user can set the priority to a negative value. **Priority** must be in the range from -40 to +40.

Returns:

0	if successful
ERR	if error

Common errors:

_priv	Only privileged users can set the priority to a negative number.

2.66 The Settime Function

function:	settime
purpose:	Set system time
user access:	privileged user
include files:	<jsysequ.h> <syslib.h>
synopsis:	int settime(time) struct sys_time *time;

Description

This function may be used by the privileged user to define the system time.

The function returns:

0	if successful
ERR	if error

Common errors:

_priv	Only a privileged user can set the system time.

2.67 The Setuser Function

function:	setuser
purpose:	Change user ID
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int setuser(idtype,idvalue,idnumber) int idtype, idvalue, idnumber;

Description

This call sets the chosen user id (**id_effective**, **id_program**, **id_login**) to a different value specified by **idvalue**:

idvalue	new id
id_effective	present effective id
id_login	user login id
id_program	program owner id
id_number	idnumber specified

The function returns:

0	if successful
ERR	if error

Common errors:

_priv	Only privileged users can set the user number to an arbitrary integer.

2.68 The Shell Function

function:	shell
purpose:	Execute a Shell process
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int shell(argv) char *argv[];

Description

The **shell** call begins execution of a shell and does not return control to the calling process. The call is similar to the **fshell** call, except that a new process is not created.

In every case **argv0** should point to the string "shell" (or "sh").

If you want to execute a program then:

```
argv[1] --> "-p"
argv[2] --> full program name
argv[3] --> arg1 of the program
argv[4] --> arg2 of the program
.....
Last pointer should be zero
```

If you want to execute a command line then

```
argv[1] -->     "-c"
argv[2] -->     command line
argv[3]         0
```

If you want to execute a command file then

```
argv[1] -->     command file name
argv[2]         0
```

or

```
argv[1] -->     "-q"
argv[2] -->     command file name
argv[3]         0
```

In the first form the commands from the command file will be echoed. In the second form, they will not be echoed.

The shell system call replaces the current code with the code of the Shell program. If an error is encountered after the original code has been scrapped, the original program quietly terminates.

This implementation of the shell system call differs in two aspects from the implementation of the shell system call in the older versions of Cromix-Plus (older than 31.11):

- The new code actually overlays the old code so that at no point do the old and the new code reside in the memory.

- Only channels **stdin**, **stdout**, and **stderr** are retained instead of all channels.

The function returns:

does not return	if no errors
ERR	if error

Common errors:

_badaddress	The address passed to the system call does not belong to the user's address space.

2.69 The Shmat Function

function:	shmat
purpose:	Attach shared memory segment
user access:	all users
include files:	<jsysequ.h> <syslib.h> <ipc.h> <shm.h>
synopsis:	char *shmat(shmid,shmflg) int shmid, shmflg;

Description

Shmat attaches the shared memory segment associated with the shared memory identifier **shmid** and returns its address.

The segment is attached for reading if (shmflg & SHM_READONLY) is "true", otherwise it is attached for reading and writing.

The function returns

address of the attached segment	if no error;
NULL	if error.

Common errors:

_badvalue	**Shmid** is not a valid shared memory identifier.
_ipcaccess	Operation permission is denied to the calling process.
_ipcspace	The system imposed limit on the number of shared memory segments, that a process can attach, has been reached.

2.70 The Shmctl Function

function: shmctl
purpose: Control operations for shared memory

user access: all users

include files: <jsysequ.h>
<syslib.h>
<ipc.h>
<shm.h>

synopsis: int shmctl(shmid,cmd,buf)
int shmid, cmd; struct shmid_ds *buf;

Description

Shmctl provides a variety of shared memory control operations as specified by **cmd**. The following commands are available:

IPC_STAT — Place the current value of each member of the data structure associated with the shared memory identifier **shmid** into the structure pointed to by **buf**.

IPC_SET — Set the values of the following members of the data structure associated with **shmid** to the corresponding values found in the **shmid_ds** structure pointed to by **buf**:

shm_perm.uid
shm_perm.gid
shm_perm.mode (low order 9 bits only)

This command can only be executed by the super user or by a process that has an effective user ID equal to the **shm_perm.uid** in the data structure associated with the **shmid**.

IPC_RMID — Remove the shared memory identifier specified by **shmid** from the system and destroy the shared memory segment and data structure associated with it. This command can only be executed by a privileged user or by the creator of the shared memory segment.

The function returns:

0 if succesful;
ERR if error.

Common errors:

_badvalue	**Shmid** is not a valid shared memory identifier.
_ipcaccess	Operation permission is denied to the calling process.
_badvalue	Invalid command.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.71 The Shmdt Function

function:	shmdt
purpose:	Detach shared memory segment
user access:	all users
include files:	<jsysequ.h> <syslib.h> <ipc.h> <shm.h>
synopsis:	int shmdt(shmp) char *shmp;

Description

Shmdt detaches from the calling process the shared memory segment located at address **shmp**.

The function will return:

0	if successful;
ERR	if error.

Common errors:

_badvalue	The shared memory pointer supplied was not obtained from the previous **_shmat** system call.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.72 The Shmget Function

function:	shmget
purpose:	Get shared memory identifier
user access:	all users
include files:	<jsysequ.h> <syslib.h> <ipc.h> <shm.h>
synopsis:	int shmget(key,size,shmflg) long key; int size, shmflg;

Description

Shmget returns the shared memory identifier associated with key.

A shared memory identifier and associated date structure and shared memory segment of **size** bytes are created for **key** if one of the following are true:

> **Key** is equal to IPC_PRIVATE.
>
> **Key** does not already have a shared memory identifier associated with it, and (shmflg & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

> **Shm_perm.cuid**, **shm_perm.uid**, **shm_perm.cgid**, **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low order nine bits of **shm_perm.mode** are set equal to the low order nine bits of **shmflg**. **Shm_segsz** is set equal to the value of size.
>
> **Shm_lpid**, **shm_nattch**, **shm_atime**, **shm_dtime** are set equal to zero.
>
> **Shm_ctime** is set equal to the current time.

The function returns:

a nonnegative shared memory identifier	if successful;
ERR	if error.

Common errors:

_ipcaccess	A shared memory identifier exists for **key** but operation permission as specified by the low order nine bits will not be granted.
_ipcnoent	A shared memory identifier does not exist for **key** and the create bit in flags is not set.
_ipcspace	There is no space in the system to create another shared memory identifier.
_ipcexists	A shared memory identifier exists for **key** but both create and exclusive bits in flags are set.
_badvalue	The size specified is either not positive, or greater than the system imposed limit, or greater than the existing size in case the shared memory identifier already exists.

2.73 The Signal Function

function:	signal
purpose:	Set up a trap to receive a signal
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	void (*signal(stype,execution_address))() int stype; void (*execution_address)();

Description

This function sets up an execution address for a signal of type **stype**. Note that the execution address is set to 1 (ignore) after the signal routine is called.

An execution address of zero means the process should abort on reception of the signal, an address of one means the process should ignore the signal.

This function is coded in assembler so that it actually installs its own trap routines which in turn call the user's trap routine. The assembler part of the trap routine takes care to save and restore all user registers before calling the user trap routine.

The user trap function is called with the signal number as its only argument.

The function returns:

Previous trap function (or zero or 1)	if successful;
ERR	if error.

Common errors:

_badvalue	Bad signal number.

2.74 The Sleep Function

function:	sleep
purpose:	Sleep a number of seconds
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int sleep(numsec) int numsec;

Description

This call puts the process to sleep for **numsec** seconds. The function returns the number of seconds the process was to sleep when a signal interrupted its dreams.

Common errors:

_ssignal	The sleep was interrupted by a signal.

2.75 The String Function

function: strcat, strncat, strcmp, strncmp, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok

purpose: String functions.

user access: all users

include files: <string.h>

synopsis:

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

char *strncat(s1,s2,n)
char *s1, *s2; int n;

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

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

char *strncpy(s1,s2,n)
char *s1, *s2; int n;

int strlen(s)
char *s;

char *strchr(s,c)
char *s, c;

char *strrchr(s,c)
char *s, c;

char *strpbrk(s1,s2)
char *s1, *s2;

int strspn(s1,s2)
char *s1, *s2;

int strcspn(s1,s2)
char *s1, *s2;

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

Description

The arguments **s1**, **s2**, and s point to strings (arrays of characters terminated by a null character). The

functions **strcat**, **strncat**, and **strncpy** all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

Strcat appends a copy of string **s2** to the end of string **s1**. **Strncat** appends at most **n** characters. Each returns a pointer to the null terminated result.

Strcmp compares its arguments and returns an integer less than, equal to, or greater than zero, according as **s1** is lexicographically less than, equal to, or greater than **s2**. **Strncmp** makes the same comparison but looks at most n characters.

Strncpy sopies string **s2** to **s1**. It copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null terminated if the length of **s2** is **n** or more. The function returns **s1**.

Strlen returns the number of characters in **s**, not including the terminating null character.

Strchr (strrchr) returns a pointer to the first (last) occurence of character **c** in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

Strspn (strcspn) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

Strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that on subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no tokens remain in **s1**, a NULL pointer is returned.

Notes

For user convenience, all these functions are declared in the optional **<string.h>** header file.

Strcmp uses native character comparison.

All string movement is performed character by character starting at the left. Thus overlapping moves towards the left will work as expected, but overlapping moves to the right may yield surprises.

2.76 The Strtol Function

function: strtol, atol, atoi
purpose: Convert string to integer

user access: all users

synopsis:
```
long strtol(str,ptr,base)
char *str, **ptr; int base;

long atol(str)
char *str;

int atoi(str)
char *str;
```

Description

Strtol returns as a long integer the value represented by the character string **str**. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters are ignored.

If the value of **ptr** is not (char **) NULL, a pointer to the character terminating the scan is returned in ***ptr**. If no integer can be formed, ***ptr** is set **str**, and zero is returned.

If base is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and 0x or 0X is ignored if base is 16.

If base is zero, the string itself determines the base thus: after an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise decimal conversion is used.

Atol(str) is equivalent to strtol(str, (char **) NULL, 10).

Atoi(str) is equivalent to (int) strtol(str, (char **) NULL, 10).

2.77 The Tgread Function

function: tgread, tgnum, tgbool, tgstr, tprint
purpose: Termcaps decoding

user access: all users

include files: none

synopsis:

```
int tgread(channel,buffer,size)
int channel;
char *buffer;
int size;

int tgnum(buffer,name)
char *buffer;
char *name;

int tgbool(buffer,name)
char *buffer;
char *name;

int tgstr(buffer,name,string,size)
char *buffer;
char *name;
char *string;
int size;

int tprint(buffer,format,arg,...)
char *buffer;
char *format;
```

Description

This set of functions is used to decode the **/etc/termcaps** (or an equivalent) file. First the **tgread** function should be called to read the description of the current terminal into buffer. With buffer succesfully filled up, functions **tgnum**, **tgbool**, and **tgstr** can be used to extract individual descriptions.

Tgread

The **tgread** function has the arguments

channel	channel number of the **termcaps** file
buffer	where the current terminal information will be stored
size	size of **buffer**

The function returns

zero	Normal termination. In this case **buffer** will contain one long zero terminated line, or an empty line if there is no description for the current terminal
nonzero	On I/O error or buffer overflow.

Tgnum

The **tgnum** function has the arguments

buffer	The buffer filled in by **tgread**.
name	The character string identifying the numeric capability.

The function returns

value	The value of the numeric capability.
-1	If the capability is not found or it is not numeric.

Tgbool

The **tgbool** function has the arguments

buffer	The buffer filled in by **tgread**.
name	The character string identifying the **Boolean** capability.

The function returns

0	If the capability is not defined.
1	If the capability is defined.
-1	If the capability is defined but not **Boolean**.

Tgstr

The **tgstr** function has the arguments

buffer	The buffer filled in by **tgread**.
name	The character string identifying the string capability.
string	Buffer where the zero terminated extracted string will be stored.
size	Size of **string**.

The function returns

length	Length of the string. The terminating zero byte is not counted.
-1	If the capability is not found or is not of string type.

Tprint

Escape sequences should be written out in one piece. The **wrseq** system call should be used to do it. In case of strings with arguments like cursor movement strings; the string to be written out must first be constructed by the **tprint** function.

The **tprint** function acts like **sprintf** function. It can be used to construct a string from the termcaps description.

The **tprint** function has the arguments:

buffer	The buffer where the string will be built.
format	The terminal capability **string**.
	Additional arguments might be needed as in the case of cursor addressing strings.

The function returns:

length	Length of the constructed string. The terminating zero byte is not counted.

2.78 The Trunc Function

function:	trunc
purpose:	Truncate file length to current position
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int trunc(channel) int channel;

Description

Trunc truncates (or extends) an open file to current file position.

Returns:

0	if successful
ERR	if error

Common errors:

_notopen	The channel is not open.

2.79 The Uchstat Function

function:	uchstat
purpose:	Change status of a process
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int uchstat(procid,type,val) int procid, type, val;

Description

Uchstat changes the component of the process table belonging to the process **procid**, to value **val**. If (**procid == 0**) it will affect the current process. Only a privileged user can change process tables other than his own.

The components that can be changed are

usr_ctty	controlling terminal device number
usr_prior	process priority
usr_term	termcaps **ident**
usr_hdevn	user home directory device number
usr_hinum	user home directory inode number
usr_cdevn	user current directory device number
usr_cinum	user current directory inode number
usr_static	user defined data pointer
usr_job	job ID

The function returns:

0	if successful
ERR	if error

Common errors:

_noproc	Such a process does not exist.
_priv	Only a privileged user can change somebody else's process table.

2.80 The Unlock Function

function:	unlock
purpose:	Unlock a locked sequence
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int unlock (lock_sequence, ltype, llength) char *lock_sequence; int ltype, llength;

Description

Unlock the locked sequence.

Returns:

0	if successful
ERR	if error

Common errors:

_badaddress	The address passed to the system call does not belong to the user's address space.

2.81 The Unmount Function

function:	unmount
purpose:	Disable access to a file system
user access:	privileged user
include files:	<jsysequ.h> <syslib.h>
synopsis:	int unmount(devpath,eject) char *devpath; int eject;

Description

Unmount the device. The argument **eject** should be

0	do not eject
1	do eject

Returns:

0	if successful
ERR	if error

Common errors:

_notmount	The file system is not mounted.
_fsbusy	The file system is in use.
_badname	Pathname to device is illegal.
_notexist	Such a device does not exist.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.82 The Update Function

function:	update
purpose:	Update all open files
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int update()

Description

This call flushes all buffers belonging to the process. The function returns:

0	if successful
ERR	if error

2.83 The Ustat Function

function:	ustat
purpose:	Get status of a process
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int ustat(procid,type) int procid, type;

Description

Ustat extracts the component of the process table belonging to the process **procid**. If (**procid == 0**) it will affect the current process. Only a privileged user can change process tables other than his own.

The components that can be accessed are

usr_ctty	controlling terminal device number
usr_prior	process priority
usr_parent	parent process ID
usr_memp	program address
usr_mems	total memory size
usr_time	process time (miliseconds)
usr_ctime	children time (miliseconds)
usr_user	process owner
usr_group	process group
usr_term	termcaps ident
usr_hdevn	user home directory device number
usr_hinum	user home directory inode number
usr_cdevn	user current directory device number
usr_cinum	user current directory inode number
usr_static	user defined data pointer
usr_job	job ID

The function returns:

requested value	if successful
ERR	if error

Common errors:

_noproc	There is no such process.
_priv	Only a privileged user can access an arbitrary process.

2.84 The Version Function

function:	version
purpose:	Get version number of operating system
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int version()

Description

This function returns the current version of the system in BCD form. The function also computes the CRC of the copy of the kernel code in memory and compares it to a stored value. If they do not match, ERR is returned and **errno** is set accordingly.

Common errors:

_corrupt	The "readonly" part of the operating system has been corrupted.

2.85 The Wait Function

function:	wait
purpose:	Wait for a child process to terminate
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int wait(flag,childid,statuses) int flag, childid, statuses[2];

Description

Wait for execution of a child process to terminate and store the child's termination status in **statuses**.

Childid is the process ID number of the process upon which the **Wait** function must wait. **If childid** is zero, the function will wait for the termination of any child process.

If **(flag&1)** is zero, the function will not return until a child terminates. If **(flag&1)** is nonzero, the function will return immediately. If there is no terminated child, the function will return an error.

When a child process terminates, its process table remains allocated so that the parent will be able to inspect its termination status. There are only two ways to get rid of the terminated child's process table:

- the parent collects its termination status by means of the **Wait** function;

- the parent process terminates, whereupon all process tables belonging to its terminated children will be assigned to process one.

If **(flag&2) is nonzero, the process table will not be discarded, in all other respects the wait** function behaves as determined by **(flag&1)**. This possibility is useful to find out when the process terminates. As the process table is not yet discarded, the **ustat** function can be used to pick up various pieces of data from it.

The function returns:

child ID and statuses	if no error
ERR	if error

Statuses are

statuses[0]	process termination status (exit value)
statuses[1]	system termination status (signal number if killed)

Common errors:

_nochild	There is no such child to wait for.

2.86 The Wrbyte Function

function:	wrbyte
purpose:	Write a byte
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int wrbyte(channel,byte) int channel, byte;

Description

Write a byte.

Returns:

0	if successful
ERR	if error

Common errors:

_notopen	The channel is not open.
_filaccess	The user does not have write access to the channel.

2.87 The Wrline Function

function:	wrline
purpose:	Write a line
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int wrline(channel,buffer) int channel; char *buffer;

Description

Write bytes from the buffer to the channel until one of the following happens:

the '\n' character is written out
the '\0' character is written out
a signal was trapped

The function returns

number of bytes written	if successful
ERR	if error

Common errors:

_notopen	The channel is not open.
_filaccess	The user does not have write access to the channel.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.88 The Wrseq Function

function:	wrseq
purpose:	Write sequential bytes
user access:	all users
include files:	<jsysequ.h> <syslib.h>
synopsis:	int wrseq(channel,buffer,bytecount) int channel, bytecount; char *buffer;

Description

Write bytes from buffer until:

bytecount bytes are written
a signal has been trapped

Return:

number of bytes written	if successful
ERR	if error

common errors:

_notopen	The channel is not open.
_filaccess	The user does not have write access to the channel.
_badaddress	The address passed to the system call does not belong to the user's address space.

2.89 The Z80to68 Function

function: dz80to68, fz80to68, iz80to68, lz80to68, uz80to68
purpose: Convert from Z80 C format to 68000 C format.

user access: all users

include files:
```
<jsysequ.h>
<syslib.h>
```

synopsis:
```
double dz80to68(p)
char *p;

double fz80to68(p)
char *p;

int iz80to68(p)
char *p;

int lz80to68(p)
char *p;

int uz80to68(p)
char *p;
```

Description

The **dz80to68** function converts data of type double from Z80 C format to 68000 C format. The pointer **p** should point to an 8-byte string holding a double value in Z80 format. The value returned is the same double value in 68000 format.

The **fz80to68** function converts data of type float from Z80 C format to 68000 C format. The pointer **p** should point to a 4-byte string holding a float value in Z80 format. The value returned is the same value (of type double) in 68000 format.

The **iz80to68** function converts data of type integer from Z80 C format to 68000 C format. The pointer **p** should point to a 2-byte string holding an integer value in Z80 format. The value returned is the same value in 68000 format.

The **lz80to68** function converts data of type long from Z80 C format to 68000 C format. The pointer **p** should point to a 4-byte string holding a long value in Z80 format. The value returned is the same value in 68000 format.

The **uz80to68** function converts data of type unsigned from Z80 C format to 68000 C format. The pointer p should point to a 2-byte string holding an unsigned value in Z80 format. The value returned is the same value in 68000 format.

Chapter 3 - Assembler System Call Summary

The system call instruction (**Jsys**) will return with the Carry bit clear if the call was successful. If the call was unsuccessful for any reason, the Carry bit will be set and the D0 register will contain the error number.

```
_alarm
        move.L      <number of seconds>, D3
        jsys        #_alarm

_boot
        lea         <address of new system>, A0
        move.L      <size>, D1
        jsys        #_boot

_caccess
        move        <channel>, D1
        move        <access bits>, D2
        jsys        #_caccess

_cchstat
        move        <channel>, D1
        move        <status type>, D2
        move        <new value>, D3
        move        <access mask>, D4       (only for access)
        lea         <buffer>, A1            (only for times)
        jsys        #_cchstat

_chdup
        move        <existing channel>, D1
        jsys        #_chdup
        move.L      D2, <duplicate channel>

_chkdev
        move        <type of device>, D2
        move        <major device number>, D3
        move        <minor device number>, D4
        jsys        #_chkdev

_clink
```

```
                move        <channel>, D1
                lea         <new pathname>, A1
                jsys        #_clink

_close
                move        <channel>, D1
                jsys        #_close

_create
                lea         <pathname>, A0
                move        <access mode>, D2
                move        <exclusive mask>, D3
                jsys        #_create
                move.L      D1, <channel>

_cstat
                move        <channel>, D1
                move        <status type>, D2
                lea         <buffer>, A1                    (if necessary)
                jsys        #_cstat <depends on status type>

_delete
                lea         <pathname>, A0
                jsys        #_delete

_divd
                move.L      <dividend>, D1
                move.L      <divisor>, D2
                jsys        #_divd
                move.L      D3,<quotient>
                move.L      D4,<remainder>

_error
                move        <error number>, D0
                move        <channel>, D1
                lea         <pathname>, A0                  (if needed)
                lea         <alternate pathname>, A1        (if needed)
                jsys        #_error

_exchg
                move        <channel number>, D1
                move        <channel number>, D2
                jsys        #_exchg

_exec
                lea         <argument list>, A1
                lea         <pathname>, A0
                jsys        #_exec
```

```
_exit
        move    <termination status>, D3
        jsys    #_exit

_faccess
        move    <access bits>, D2
        lea     <pathname>, A0
        jsys    #_faccess

_fchstat
        lea     <pathname>, A0
        move    <status type>, D2
        move    <new value>, D3
        move    <access mask>, D4       (only for access)
        lea     <buffer>, A1            (only for times)
        jsys    #_fchstat

_fexec
        lea     <argument list>, A1
        lea     <pathname>, A0
        move    <signal mask>, D1
        move    <signal values>, D2
        jsys    #_fexec
        move.L  D3, <new PID>

_flink
        lea     <old pathname>, A0
        lea     <new pathname>, A1
        jsys    #_flink

_fshell
        lea     <argument list>, A1
        move    <signal mask>, D1
        move    <signal values>, D2
        jsys    #_fshell
        move.L  D3, <new PID>

_fstat
        lea     <pathname>, A0
        move    <status type>, D2
        lea     <buffer>, A1            (if necessary)
        jsys    #_fstat

_getdate
        jsys    #_getdate
        move.L  D0, <weekday>
        move.L  D1, <year>
        move.L  D2, <month>
        move.L  D3, <day>
```

```
_getdir
        lea     <buffer>, A0
        jsys    #_getdir

_getgroup
        move    <id type>, D2
        jsys    #_getgroup
        move.L  D3, <group number requested>

_getmode
        move    <channel>, D1
        move    <mode type>, D2
        jsys    #_getmode
        move.L  D3, <mode value>

_getpos
        move    <channel number>, D1
        jsys    #_getpos
        move.L  D3, <file position>

_getprior
        jsys    #_getprior
        move.L  D3, <process priority>

_getproc
        jsys    #_getproc
        move.L  D3, <PID>

_gettime
        jsys    #_gettime
        move.L  D1, <hour>
        move.L  D2, <minute>
        move.L  D3, <second>

_getuser
        move    <id type>, D2
        jsys    #_getuser
        move.L  D3, <user>

_indirect
        move    <call number>, D0
        ;all other registers as required by the call
        jsys    #_indirect

_kill
        move    <signal type>, D2
        move    <process id>, D3
        jsys    #_kill
```

```
_lock
        move    <lock type>, D2
        move    <lock length>, D3
        lea     <lock sequence>, A0
        jsys    #_lock

_makdev
        move    <type of device>, D2
        move    <major device #>, D3
        move    <minor device #>, D4
        lea     <pathname>, A0
        jsys    #_makdev

_makdir
        lea     <pathname>, A0
        jsys    #_makdir

_memory
        move.L  <mask>, D1              (if allocating)
        move    <type>, D2
        move.L  <size>, D3
        lea     <memory pointer>, A0    (if deallocating)
        jsys    #_memory
        move.L  A0, <memory pointer>    (if allocating)

_mount
        move    <type of access>, D2
        lea     <dummy pathname>, A0
        lea     <block device pathname>, A1
        jsys    #_mount

_msgctl
        move.L  <msqid>, D1
        move.L  <command>, D2
        lea     <buffer>, A0
        jsys    #_msgctl

_msgget
        move.L  <key>, D1
        move.L  <msgflg>, D2
        jsys    #_msgget
        move.L  D3, <msqid>

_msgrcv
        move.L  <msqid>, D1
        move.L  <msgflg>, D2
        move.L  <msgsz>, D3
        move.L  <msgtyp>, D4
        lea     <message>,A0
```

```
                jsys        #_msgrcv
                move.L      D3, <msgsz>

_msgsnd
                move.L      <msqid>, D1
                move.L      <msgflg>, D2
                move.L      <msgsz>, D3
                lea         <message>,A0
                jsys        #_msgsnd

_mult
                move.L      <multiplicand>, D1
                move.L      <multiplicator>, D2
                jsys        #_mult
                move.L      D3,<product>

_open
                lea         <pathname>, A0
                move        <access mode>, D2
                move        <exclusive mask>, D3
                jsys        #_open
                move.L      D1, <channel>

_pause
                jsys        #_pause

_phys
                lea         <addr>, A0
                move.L      #<size>, D3
                move        #<access>, D2
                jsys        #_phys

_pipe
                jsys        #_pipe
                move.l      D1, <reading side>
                move.l      D2, <writing size>

_printf
                move        <channel>, D1
                lea         <control string>, A0
                ;push all arguments, last argument first
                jsys        #_printf
                ;pop all arguments

_ptrace
                move        <function code>, D1
                move        <pid>, D2
                lea         <address>, A0
                lea         <data>, A1
```

```
                move.L      <count>, D3
                jsys        #_ptrace

_rdbyte
                move        <channel>, D1
                jsys        #_rdbyte
                move.L      D0, <value read>

_rdline
                move        <channel>, D1
                move.L      <maximum bytes>, D3
                lea         <buffer>, A0
                jsys        #_rdline
                move.L      D3, <bytes read>

_rdseq
                move        <channel>, D1
                move.L      <byte count>, D3
                lea         <buffer>, A0
                jsys        #_rdseq
                move.L      D3, <bytes read>

_semctl
                move.L      <semid>, D1
                move.L      <command>, D2
                move.L      <semnum>,D4
                move.L      <arg>, D3
                jsys        #_semctl
                move.L      D3,<return value)

_semget
                move.L      <key>, D1
                move.L      <semflg>, D2
                move.L      <nsems>, D4
                jsys        #_semget
                move.L      D3, <semid>

_semop
                move.L      <semid>, D1
                move.L      <nsops>, D2
                move.L      <sembuf>, A0
                jsys        #_semop

_setdate
                move        <year>, D1
                move        <month>, D2
                move        <day of the month>, D3
                jsys        #_setdate
```

```
_setdir
        lea     <buffer>, A0
        jsys    #_setdir

_setgroup
        move    <type of id to change>, D1
        move    <new id type>, D2
        move    <new id number>, D3
        jsys    #_setgroup

_setlev
        move.L  <interrupt level>,D1
        jsys    #_setlev

_setmode
        move    <channel>, D1
        move    <mode type>, D2
        move.L  <new value>, D3
        move    <mask>, D4
        jsys    #_setmode
        move.L  D3, <old value>

_setpos
        move    <channel number>, D1
        move    <mode>, D2
        move.L  <file pointer>, D3
        jsys    #_setpos

_setprior
        move    <priority number>, D3
        jsys    #_setprior

_settime
        move    <hours>, D1
        move    <minutes>, D2
        move    <seconds>, D3
        jsys    #_settime

_setuser
        move    <type of id to change>, D1
        move    <new id type>, D2
        move    <new id number>, D3
        jsys    #_setuser

_shell
        lea     <argument list>, A1
        jsys    #_fexec

_shmat
```

```
                move.L      <shmid>, D1
                move.L      <shmflg>, D2
                jsys        #_shmat
                move.L      A0, <memptr>

_shmctl
                move.L      <shmid>, D1
                move.L      <command>, D2
                lea         <buffer>, A0
                jsys        #_shmctl

_shmdt
                move.L      <memptr>, A0
                jsys        #_shmdt

_shmget
                move.L      <key>, D1
                move.L      <shmflg>, D2
                move.L      <size>,D4
                jsys        #_shmget
                move.L      D3, <shmid>

_signal
                move        <type of signal>, D2
                lea         <execution address>, A0
                jsys        #_signal
                move.L      A0, <old trap address>

_sleep
                move.L      <number of seconds to sleep>,D3
                jsys        #_sleep
                move.L      D3,<number of seconds left>

_trunc
                move        <channel>, D1
                jsys        #_trunc

_uchstat
                move        <process id>, D1
                move        <status type>, D2
                move        <new value>, D3
                jsys        #_uchstat

_unlock
                move        <lock type>, D2
                move        <lock length>, D3
                lea         <lock sequence>, A0
                jsys        #_unlock
```

```
_unmount
        move    <eject flag>, D2
        lea     <block device pathname>, A0
        jsys    #_unmount

_update
        jsys    #_update

_ustat
        move    <process id>, D1
        move    <status type>, D2
        jsys    #_ustat
        move.L  D3,<status value>

_version
        jsys    #_version
        move.L  D3, <version number>

_wait
        move    <conditional flag>, D1
        move    <process ID>, D3
        jsys    #_wait
        move.L  D3, <child PID>
        move.L  D2, <termination status>
        move.L  D1, <signal number >

_wrbyte
        move    <channel>, D1
        move.B  <byte>, D0
        jsys    #_wrbyte

_wrline
        move    <channel>, D1
        lea     <buffer>, A0
        jsys    #_wrline
        move.L  D3, <bytes written>

_wrseq
        move    <channel>, D1
        move.L  <byte count>, D3
        lea     <buffer>, A0
        jsys    #_wrseq
        move.L  D3, <bytes written>
```

Chapter 4 - Disk Allocation Under Cromix-Plus

This chapter describes disk allocation under the Cromix Operating System. Any small or large floppy disk or hard disk formatted for use under the Cromix system is divided into three major sections: the **System Area**, **Inode Area**, and **Data Area**. These disks are formatted with a block size of 512 bytes (decimal).

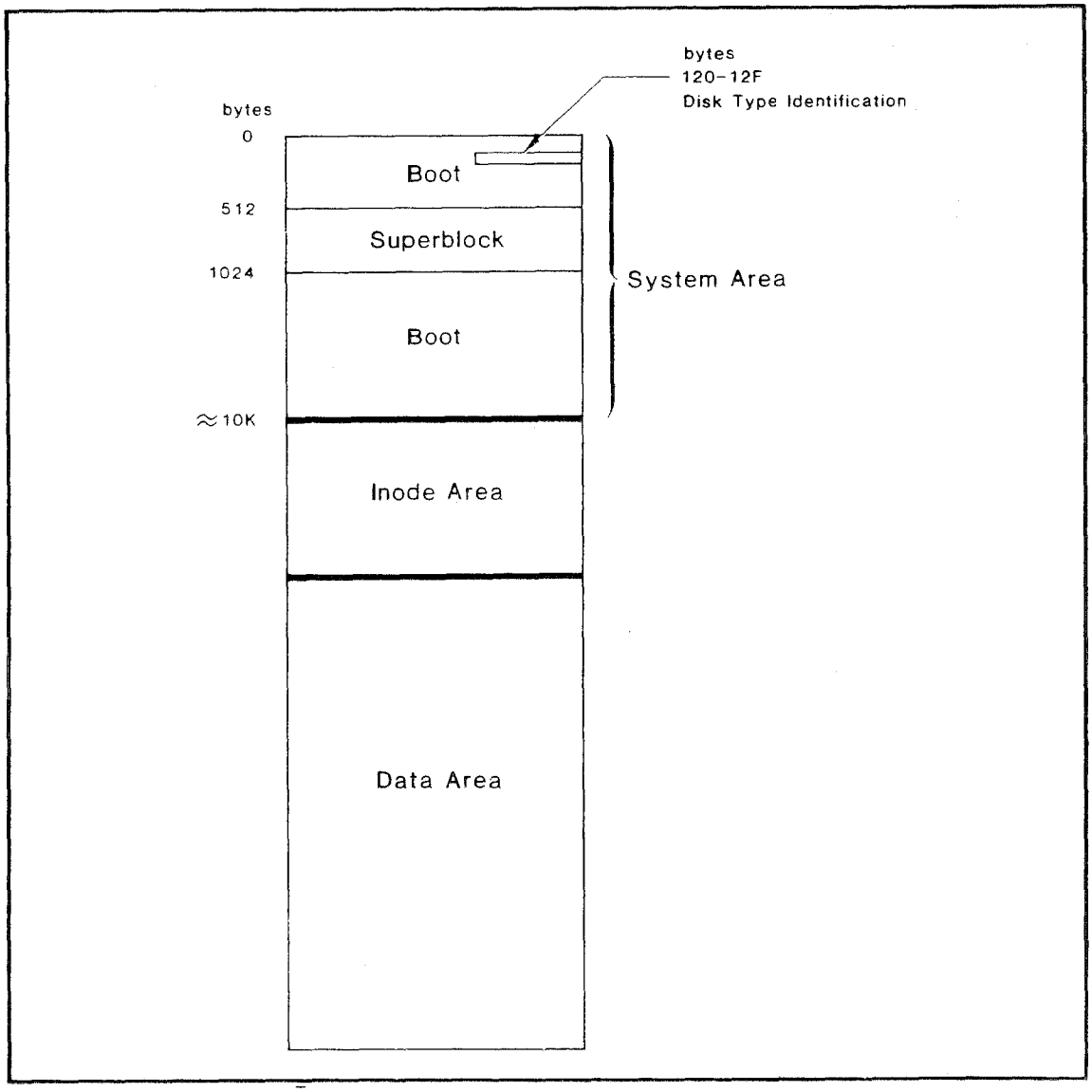

Figure 4-1: LAYOUT OF A CROMIX DISK

4.1 System Area

The System Area has a default size of 10K bytes for all disk types. Although it is not recommended, the size of this area can be specified when running the **Makfs** (make file system) utility program.

The System Area contains system information required for booting up (boot tracks) and disk type identification. In addition, it contains the Superblock, and, for hard disks, the alternate track table and the partition table.

4.2 Disk Type Identification

On Cromix-format floppy disks, bytes 120 through 127 (in the first block) contain ASCII-encoded data detailing the type and use of the disk.

Floppy disks have six letters in this position. When formatted for use with the Cromix Operating System, byte 120 contains a **C**. Byte 121 contains an **S** or **L**, to indicate a Small (5") or Large (8") floppy disk. Bytes 122-123 contain the characters **SS** or **DS**, indicating a Single Sided or Double Sided Disk. Bytes 124-125 contain the characters **SD** or **DD**, indicating a Single Density or Double Density disk. Bytes 126-127 are not significant, but are reserved for future use.

Cromix-Plus also supports uniform-format floppy disks, which contain no identification information in the first block. In uniform format, all tracks are the same. All sectors are the same size: the sector size might be 128, 256, or 512 bytes.

On hard disks, bytes 68h through 7Fh contain disk type identification. The following table details this area of the disk.

68-69	Number of cylinders, not counting alternate tracks (2 bytes)
6A-6B	Number of alternate tracks (2 bytes)
6C	Number of surfaces (1 byte)
6D	Number of sectors per track (1 byte)
6E-6F	Number of bytes per sector (2 bytes)
70-71	Byte count of start of alternate track table (2 bytes)
72-73	Cylinder number of start of disk (2 bytes)
74-75	Cylinder number where alternate tracks are located (2 bytes)
76-77	Byte count of start of partition table (2 bytes)
78-7B	Hard disk identifier, usually CSTD (4 bytes)
7C-7D	Cylinder number where write precompensation starts
7E-7F	Reserved for future use (4 bytes)

4.3 Superblock

The second block (bytes 512-1023) is the Superblock. This block contains housekeeping information for the disk, including the **Block Free List** and the **Inode Free List**.

The **Block Free List** (sometimes called the Free List) is a stack of 80 4-byte pointers, preceded by a 2-byte counter. Each pointer in the **Block Free List** points to a disk block not in use. As information is deleted from the disk, the Block Free List grows; as information is written to the disk, it shrinks.

The last pointer used (actually, the first pointer in the list) points to a block on the disk that contains another **Block Free List**. When the **Block Free List** in the Superblock is exhausted, the next **Block Free List** is loaded into the Superblock. When the **Block Free List** in the Superblock is full, it is moved to the Data Area of the disk.

The **Inode Free List** is a stack of 80 2-byte inode numbers preceded by a 2-byte counter. Each entry in the **Inode Free List** is the number of an unused inode. When this stack is exhausted, the Cromix system searches through the inode table and replenishes the stack with the numbers of additional inodes not in use.

4.4 Alternate Track Table

The Alternate Track Table for the hard disk is located at the top of the System Area, before the Inode Area.

4.5 Inode Area

An inode is a descriptor for one file; it contains a collection of information pertaining to the file.

The first 48 bytes contain information on the number of links to the file, allowable access modes, and most recent access times for various types of access.

The last 80 bytes of the inode contain 4-byte pointers to the file itself. The first 16 of these pointers each points to a block of the file. The first pointer points to the first block (bytes 0-511); the second pointer points to the second block (bytes 512-1023), and so on. This continues until the whole file has been pointed to, or until the sixteenth pointer has been used (pointing to bytes 7680-8191). Thus, if the file is 8 Kbytes or smaller, only the first 16 (or fewer) pointers need be used.

If the file described by the inode is larger than 8 Kbytes, the seventeenth pointer is used. This pointer points to a block of 128 pointers. Each of these pointers points to a block of the file in a manner similar to the first 16 pointers described above. Thus the seventeenth pointer describes the next 64 Kbytes of the file.

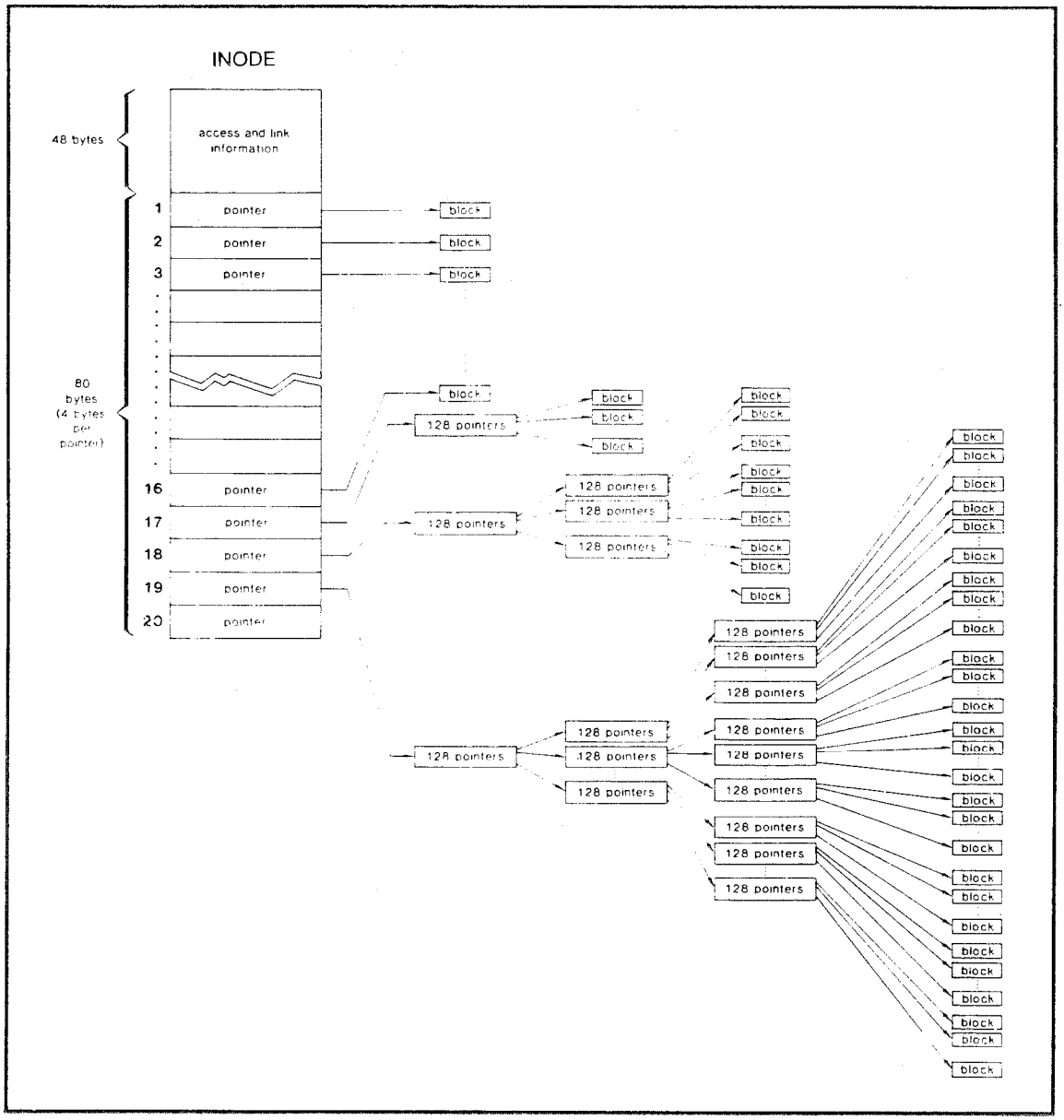

Figure 4-2: INODE LAYOUT

If the file is larger than 72 Kbytes, the eighteenth pointer is used. This pointer points to a block of 128 pointers. Each of these points to a block of 128 pointers. These pointers, in turn, point to a block in the file. Thus, the eighteenth pointer describes the next 8192 Kbytes of the file. The nineteenth pointer extends one more level, covering the next 1,048,576 Kbytes of the file.

4.6 Data Area

The Data Area occupies most of the disk. All data on the disk is stored in the data area. All blocks pointed to by inodes are in this area.

Appendix A - Z80 System Calls

A.1 Summary of Z80 System Calls

alarm:

HL	= Number of seconds
jsys	.alarm

caccess:

B	= Channel number
C	= Mask
Jsys	.caccess

cchstat:

B	= Channel number
C	= st_owner, st_group
DE	= Value
Jsys	.cchstat

or

B	= Channel number
C	= st_aowner, st_agroup, st_aother
D	= Value
Jsys	.cchstat

or

B	= Channel number
C	= st_tcreate, st_tmodify, st_taccess, or st_tdumped
DE	= Point to 6 byte buffer
Jsys	.cchstat

chdup:

B	= Channel number
Jsys	.chdup
C	= New channel number

chkdev:

C	= Device type

```
        D       = Major device number
        E       = Minor device number
        Jsys    .chkdev

clink:
        B       = Channel number
        DE      = New pathname
        Jsys    .clink

close:
        B       = Channel number
        Jsys    .close

create:
        C       = Mode
        D       = Exclusive mask
        HL      = Pathname
        Jsys    .create
        B       = Channel number

cstat:
        B       = Channel number
        C       = st_owner, st_group, st_nlinks, or st_inum
        Jsys    .cstat
        DE      = Value

        or

        B       = Channel number
        C       = st_aowner, st_agroup, st_aother, or st_ftype
        Jsys    .cstat
        D       = Value

        or

        B       = Channel number
        C       = st_size
        Jsys    .cstat
        DEHL    = Value

        or

        B       = Channel number
        C       = st_devno, st_device, or st_pdevno
        Jsys    .cstat
        D       = Major device number
        E       = Minor device number

        or
```

```
        B       = Channel number
        C       = st_all
        DE      = Point to 128 byte buffer
        Jsys    .cstat

        or

        B       = Channel number
        C       = st_tcreate, st_tmodify, st_taccess, or st_tdumped
        DE      = Point to 6 byte buffer
        Jsys    .cstat

delete:
        HL      = Pathname
        Jsys    .delete

divd:
        DEHL    = Dividend
        BC      = Divisor
        Jsys    .divd
        DE      = Remainder
        HL      = Quotient

error:
        B       = Channel number
        DE      = Point to alternate pathname
        HL      = Point to pathname
        Jsys    .error

exchg:
        B       = Channel number
        C       = Channel number
        Jsys    .exchg

exec:
        DE      = Argv vector
        HL      = Pathname
        Jsys    .exec

exit:
        HL      = Exit status
        Jsys    .exit

faccess:
        C       = Mask
        HL      = Pathname
        Jsys    .faccess

fchstat:
```

```
        C       = st_owner, st_group
        DE      = Value
        HL      = Pathname
        Jsys    .fchstat

        or

        C       = st_aowner, st_agroup, st_aother
        D       = Value
        HL      = Pathname
        Jsys    .fchstat

        or

        C       = st_tcreate, st_tmodify, st_taccess, or st_tdumped
        DE      = Point to 6 byte buffer
        HL      = Pathname
        Jsys    .fchstat

fexec:
        B       = Signal mask
        C       = Signal values
        DE      = Argv vector
        HL      = Pathname
        Jsys    .fexec
        HL      = PID

flink:
        DE      = New pathname
        HL      = Pathname
        Jsys    .flink

fshell:
        B       = Signal mask
        C       = Signal values
        DE      = Argv vector
        Jsys    .fshell
        HL      = PID

fstat:
        C       = st_owner, st_group, st_nlinks, or st_inum
        HL      = Pathname
        Jsys    .fstat
        DE      = Value

        or

        C       = st_aowner, st_agroup, st_aother, or st_ftype
        HL      = Pathname
```

```
        Jsys    .fstat
        D       = Value

        or

        C       = st_size
        HL      = Pathname
        Jsys    .fstat
        DEHL    = Value

        or

        C       = st_devno, st_device, or st_pdevno
        HL      = Pathname
        Jsys    .fstat
        D       = Major device number
        E       = Minor device number

        or

        C       = st_all
        DE      = Point to 128 byte buffer
        HL      = Pathname
        Jsys    .fstat

        or

        C       = st_tcreate, st_tmodify, st_taccess, or st_tdumped
        DE      = Point to 6 byte buffer
        HL      = Pathname
        Jsys    .fstat

getdate:
        Jsys    .getdate
        D       = Day of the week
        E       = Year
        H       = Month
        L       = Day of the month

getdir:
        HL      = Buffer
        Jsys    .makdev

getgroup:
        C       = Type
        Jsys    .getgroup
        HL      = Group number

getmode:
```

```
        B       = Channel number
        C       = Mode number
        Jsys    .getmode
        D       = Mode value

        or

        DE      = Mode value

        or

        DEHL    = Mode value

getpos:
        B       = Channel number
        Jsys    .getpos
        DEHL    = File position

getprior:
        Jsys    .getprior
        HL      = Priority

getproc:
        Jsys    .getproc
        HL      = PID

gettime:
        Jsys    .gettime
        E       = Hours
        H       = Minutes
        L       = Seconds

getuser:
        C       = Type
        Jsys    .getuser
        HL      = User number

kill:
        C       = Signal number
        HL      = PID
        Jsys    .kill

lock:
        C       = Lock type
        DE      = Length of lock sequence
        HL      = Lock sequence
        Jsys    .lock

makdev:
```

```
        C       = Device type
        D       = Major device number
        E       = Minor device number
        HL      = Pathname
        Jsys    .makdev

makdir:
        HL      = Pathname
        Jsys    .makdir

memove:
        BC      = Flag (0 = read, 1 = write)
        DE      = Size of move
        HL      = Local address
        DEHL'   = Global address
        Jsys    .memove

msgget:
        DE      = Flags
        DEHL'   = Message key
        Jsys    .msgget
        BC      = Message queue identifier

msgrcv:
        BC      = Message queue identifier
        DE      = Message size
        HL      = Message buffer
        BC'     = Message flags
        DEHL'   = Type of message
        Jsys    .msgrcv
        DE      = Actual message size

msgsnd:
        BC      = Message queue identifier
        DE      = Message size
        HL      = Message buffer
        BC'     = Message flags
        Jsys    .msgsnd

mount:
        C       = Read-only flag
        DE      = Dummy pathname
        HL      = Device pathname
        Jsys    .mount

mult:
        BC      = Multiplicator
        HL      = Multiplicand
        Jsys    .mult
```

```
        DEHL    = Product

open:
        C       = Mode
        D       = Exclusive mask
        HL      = Pathname
        Jsys    .open
        B       = Channel number

pause:
        Jsys    .pause

pipe:
        Jsys    .pipe
        B       = Read channel
        C       = Write channel

printf:
        B       = Channel number
        HL      = Point to format string
                  Push    Arguments
                  Jsys    .printf
                  Pop     Arguments

rdbyte:
        B       = Channel number
        Jsys    .rdbyte
        A       = byte

rdline:
        B       = Channel number
        DE      = Line size
        HL      = Buffer
        Jsys    .rdline
        DE      = Number of bytes read

rdseq:
        B       = Channel number
        DE      = Number of bytes
        HL      = Buffer
        Jsys    .rdseq
        DE      = Number of bytes read

semget:
        BC      = Flags
        DE      = Number of semaphores
        DEHL'   = Semaphore key
        Jsys    .semget
        BC      = Semaphore group identifier
```

semop:

BC	= Semaphore group identifier
DE	= Number of operations
HL	= Sembuf pointer
Jsys	.semop

setdate:

E	= Year
H	= Month
L	= Day of the month
Jsys	.setdate

setdir:

HL	= Buffer
Jsys	.setdir

setgroup:

B	= Destination type
C	= Source type
HL	= Group number (if source type is id.hl)
Jsys	.setgroup

setmode:

B	= Channel number
C	= Mode number
D	= Mode value
E	= Mode mask
or	
D	= Mode value
or	
DE	= Mode value
or	
DEHL	= Mode value
Jsys	.setmode
D	= Mode value
or	
DE	= Mode value
or	
DEHL	= Mode value

setpos:

B	= Channel number
C	= Mode
DEHL	= Offset
Jsys	.setpos

setprior:

L	= Priority
Jsys	.setprior

settime:

E	= Hours
H	= Minutes
L	= Seconds
Jsys	.settime

setuser:

B	= Destination type
C	= Source type
HL	= User number (if source type is id.hl)
Jsys	.setuser

shell:

DE	= Argv vector
Jsys	.shell

shmat:

BC	= Shared memory identifier
DE	= Flags
Jsys	.shmat
DEHL'	= Shared memory pointer

shmdt:

DEHL'	= Shared memory pointer
Jsys	.shmdt

shmget:

BC	= Flags
DEHL	= Size of shared memory
DEHL'	= Shared memory key
Jsys	.shmget
BC	= Shared memory identifier

signal:

C	= Signal type
HL	= Trap address
Jsys	.signal
HL	= Old trap address

sleep:

HL	= Number of seconds
Jsys	.sleep
HL	= Number of seconds left

trunc:

B	= Channel number
Jsys	.trunc

unlock:

C	= Lock type
DE	= Length of lock sequence
HL	= Lock sequence
Jsys	.unlock

unmount:

C	= Eject flag
HL	= Device pathname
Jsys	.unmount

update:

Jsys	.update

version:

Jsys	.version
HL	= Version number

wait:

C	= Flag
HL	= PID
Jsys	.wait
C	= Signal number
DE	= Exit status
HL	= PID

wrbyte:

A	= Byte
B	= Channel number
Jsys	.wrbyte

wrline:

B	= Channel number
HL	= Buffer
Jsys	.wrline
DE	= Number of bytes written

wrseq:

B	= Channel number
DE	= Number of bytes

HL	= Buffer
Jsys	.wrseq
DE	= Number of bytes written

Appendix B - ASCII Character Codes

HEX	CHARACTER	HEX	CHAR	HEX	CHAR	HEX	CHAR
00h	NUL (CONTROL-@)	20h	SPACE	40h	@	60h	'
01h	SOH (CONTROL-A)	21h	!	41h	A	61h	a
02h	STX (CONTROL-B)	22h	"	42h	B	62h	b
03h	ETX (CONTROL-C)	23h	#	43h	C	63h	c
04h	EOT (CONTROL-D)	24h	$	44h	D	64h	d
05h	ENQ (CONTROL-E)	25h	%	45h	E	65h	e
06h	ACK (CONTROL-F)	26h	&	46h	F	66h	f
07h	BEL (CONTROL-G)	27h	'	47h	G	67h	g
08h	BS (CONTROL-H)	28h	(	48h	H	68h	h
09h	HT (CONTROL-I)	29h	)	49h	I	69h	i
0Ah	LF (CONTROL-J)	2Ah	*	4Ah	J	6Ah	j
0Bh	VT (CONTROL-K)	2Bh	+	4Bh	K	6Bh	k
0Ch	FF (CONTROL-L)	2Ch	,	4Ch	L	6Ch	l
0Dh	CR (CONTROL-M)	2Dh	-	4Dh	M	6Dh	m
0Eh	SO (CONTROL-N)	2Eh	.	4Eh	N	6Eh	n
0Fh	SI (CONTROL-O)	2Fh	/	4Fh	O	6Fh	o
10h	DLE (CONTROL-P)	30h	0	50h	P	70h	p
11h	DC1 (CONTROL-Q)	31h	1	51h	Q	71h	q
12h	DC2 (CONTROL-R)	32h	2	52h	R	72h	r
13h	DC3 (CONTROL-S)	33h	3	53h	S	73h	s
14h	DC4 (CONTROL-T)	34h	4	54h	T	74h	t
15h	NAK (CONTROL-U)	35h	5	55h	U	75h	u
16h	SYN (CONTROL-V)	36h	6	56h	V	76h	v
17h	ETB (CONTROL-W)	37h	7	57h	W	77h	w
18h	CAN (CONTROL-X)	38h	8	58h	X	78h	x
19h	EM (CONTROL-Y)	39h	9	59h	Y	79h	y
1Ah	SUB (CONTROL-Z)	3Ah	:	5Ah	Z	7Ah	z
1Bh	ESC (CONTROL-[)	3Bh	;	5Bh	[	7Bh	{
1Ch	FS (CONTROL-\)	3Ch	<	5Ch	\	7Ch	|
1Dh	GS (CONTROL-])	3Dh	=	5Dh	]	7Dh	}
1Eh	RS (CONTROL-^)	3Eh	>	5Eh	^	7Eh	~
1Fh	US (CONTROL-_)	3Fh	?	5Fh	_	7Fh	DEL

NUL = null
SOH = start of heading
STX = start of text
ETX = end of text
EOT = end of transmission
ENQ = enquiry
ACK = acknowledge
BEL = bell
BS = backspace
HT = horizontal tab
LF = line feed
VT = vertical tab
FF = form feed
CR = carriage return
SO = shift out
SI = shift in
DLE = data link escape

DC1 = device control 1
DC2 = device control 2
DC3 = device control 3
DC4 = device control 4
NAK = negative acknowledge
SYN = synchronous idle
ETB = end transmission block
CAN = cancel
EM = end of medium
SUB = substitute
ESC = escape
FS = file separator
GS = group separator
RS = record separator
US = unit separator
SP = space
DEL = delete

Cromemco®

280 Bernardo Ave.
P.O. Box 7400
Mountain View, CA 94039