1616: Technical Reference Manual

Version 4.066
August, 1993

Applix 1616 microcomputer project
Applix pty ltd

1616 Technical Reference Manual

The original version of this manual was written by Andrew Morton, who also wrote the entire operating system.
Additional introductory and tutorial material by Eric Lindsay
Editorial and design consultant: Jean Hollis Weber

Comments about this manual or the software it describes should be sent to:

Applix Pty Limited
Lot 1, Kent Street,
Yerrinbool, 2575
N.S.W. Australia
(048) 839 372

Private BBS systems (ringback) on (02) 554 3114 and (02) 540 3595

ISBN 0 947341 02 1

1
Relocatable Code Format

If a computer system is to retain multiple programs in memory simultaneously, it is necessary for the operating system to be able to load programs in at varying addresses. If a computer includes memory management hardware, most of this can be done by the hardware. Like most low cost computers, the Applix does not have the necessary hardware. In the absence of memory management hardware, the most obvious method is to write relocatable code, such as is done in the Macintosh. However this can be restrictive to programmers.

The Applix uses instead a relocating loader. Because one does not know what combination of programs is going to be in memory at any one time, you can not give each program a different load address. With relocatable program files, the operating system can load a file in at any address in memory, and then alter all the self-references within the program so that it will run correctly at the loaded address. Utilities are provided to change a program with absolute addresses into the relocatable format described later.

First, some terminology from UNIX and the C language: for our purposes a program consists of three sections: **text**, **data** and **bss**.

text	The text section of a program is the actual machine instructions: the opcodes and their operands. The text section may include some data such as strings of text, jump tables, etc.
data	More formally initialised data, the data section holds such things as constant strings, tables of numbers, initialised data structures, etc.
bss	This stands for the block storage segment. The bss is a program's variable storage area. This is where variables and data are written, stored and read.

In a program file on disk only the **text** and **data** sections need be stored. The **bss** section contains no useful information before the program has commenced execution. When 1616/OS loads an `.xrel` file into memory for execution, it fills the program's **bss** area with zeroes before the program starts executing.

For example, in the 1616/OS ROMs, the **text** and **data** sections are in the ROMs, while the **bss** section is in RAM at $400.

A relocatable format file (an `.xrel` file) contains:

- a header which contains information about the file, and about its three sections.
- the text and data itself
- relocation information with which the loader can make the code executable at whatever address it is loaded.

This relocation format only supports the relocation of long words. Short mode (16 bit or 8 bit) addressing is not supported.

The header is as follows:

Offset into file	Size	Name	Usage
0	word	magic1	Magic pattern ($601a) (actually bra 26)
2	long	textlen	Length of text section
6	long	datalen	Length of data section
10	long	bsslen	Length of bss section
14	long	symtablen	Length of symbol table
18	long		Program stack allocation
22	long	textbegin	Start address of text
26	word	relocflag	If non-zero, relocation info is included in file

The symtablen field indicates the length of the file symbol table at the end of the file. This may be set to zero if the symbol table does not exist. The textbegin field is the address at which the text segment will run without relocation. This is used for offsetting self references within the program for loading at the new address.

When the 1616/OS relocating loader loads in a relocatable (`.xrel`) file for execution it places the **text**, **data** and **bss** sections contiguously in memory, in that order.

Immediately after the relocatable file's header comes the program text, then the initialised data.

After this comes the symbol table, and then the relocation table. The relocation information consists of a coded series of pointers into the text and data sections which tells the loader which longwords need relocation. The coding scheme is as follows:

The relocation information starts with a long word, which is the offset into the loaded code of the first relocatable longword. If this offset is zero, then no relocations exist: the file contains no absolute self-references.

After the first longword comes a series of bytes. Each byte is added to the current pointer to get a pointer to the next longword which must be relocated. This will result in all the byte offsets being even numbers. The byte $01 has a special meaning: add 254 (decimal) to the location pointer without performing a relocation. This is for cases where two neighbouring relocatable longwords are more than 254 bytes apart. The entire sequence is terminated by a byte of $00.

The ***loadrel*** and ***floadrel*** system calls (.69) may be used to load and relocate `.xrel` files.

Generating .xrel files - genreloc

At present, the 1616 assembler SSASM does not produce `.xrel` files directly. The `genreloc.xrel` utility has been provided for this purpose. This program generates an `.xrel` file from two `exec` files (such as those produced by `ssasm`) whose start addresses differ by $20002.

Use this program as follows:

```
genreloc ifile.1 ifile.2 outfile.xrel bsssize
```

Where `ifile.1` is the code executable at $10002 and `ifile.2` is the code executable at $30004. `outfile.xrel` is the output file. `bsssize` is an optional argument which specifies how much **bss** space there is to be allocated at the end of the data section at run time. Note that this takes no disk space.

If the filename extensions are omitted they default to '`.1`', '`.2`' and '`.xrel`'.

When using `ssasm` to generate `ifile.1` and `ifile.2` be careful to not assemble either one at a start address of less than $10000. This is because the assembler optimises some addressing to absolute short mode, which could result in programs which start at different addresses differing in more than just their self-referential addressing.

The shell file `makexrel.shell` automates all of this: be sure not to have any .ORG pseudo-ops in the source code when using `makexrel.shell`, because this shell file adds its own.

Assembling your program twice is, of course, a nuisance. Unless you are developing a memory resident driver (MRD) the code can be tested out as an `.exec` file (requiring only one assembly) and then converted to an `.xrel` file when the code is stable.

The `relcc` program available to owners of the *HiTech C Compiler* can be used to directly generate `.xrel` programs from C code

Loading executable files into memory

The ***floadrel*** system call loads programs (`.exec` or `.xrel`) into memory. This call should be used if you wish to load code from a disk file without using the ***exec*** system calls. The loaded program can remain resident in memory after the program which loaded it terminates, provided the loaded code resides in mode 1 allocated memory.

Loadrel will load the program from a previously opened file into a specific memory location, rather than being allocated memory by the system.

Load a program - floadrel

floadrel(path, memmode)

d0	69	
d1	path	Pointer to pathname of program
d2	memmode	Memory allocation mode
Return	Error code or load address.	

It is passed path, a pointer to a null-terminated string which is the pathname of the file to load. If the pathname does not end with `.exec` or `.xrel` an error code is returned.

If path is a `.exec` file the following happens:
The start address and length of the file are read
Memory is allocated for the file using ***getfmem***
The file is loaded
The start address is returned

If path is a `.xrel` file the following happens:
The length of the file (text, data, bss) is read
Memory is allocated for the file using ***getmem***
The program is loaded in and relocated
The start address is returned

The memmode argument to this system call is the mode which is passed to ***getfmem*** or ***getmem*** when storage is allocated for the program to be loaded. See the ***getmem*** documentation for details. memmode should be 0 if the loaded program does not need to remain in memory after the currently running program has ended.

If the loaded program is to remain in memory after the currently running program has ended, use memmode = 1. Note that this leaves the loaded program safely in memory until its memory is freed or until the system is reset.

It is the caller's responsibility to release the memory which is allocated when the program is loaded in (by using ***freemem***).

Relocating loader - loadrel

loadrel(handle, addr)

d0	11	
d1	handle	Input file handle
d2	addr	Target address
Return	Error code	

Loads relocatable code from the previously opened file (see syscall 105 ***open*** to open a file) whose handle is handle. The file should be open for reading, with the file pointer positioned at a relocatable file header. The code is read in, then the relocation information is read, and the relocation is performed. Upon return the code has been loaded at (addr), and the file pointer is located at the next byte beyond the end of the relocation information.

A negative return indicates an error.

If handle = -1 then the start address of the most recently loaded program (loaded by an ***exec***) is returned. See ***floadrel*** (syscall 69) above for loading a program.

2
Memory Resident Drivers

A memory resident driver (MRD) is a program which permanently resides in memory. Memory resident drivers are read into system memory from the file `mrdrivers` on /F0, /F1, /H0 or /H1 during level 0 reset processing . The `mrdrivers` file is created by linking together zero, one or more memory resident driver program (.mrd) files with the buildmrd program. The `mrdrivers`file also contains system configuration information.

As the `mrdrivers` file is read from magnetic disk at boot time, the users of tape-only systems will not be able to use memory resident drivers.

Memory resident drivers may be used for background interrupt processing, pop-up applications, alteration of existing system calls and 1616/OS inbuilt commands, etc.

Memory resident drivers may be written as memory-resident transient programs: they are called when their identifying name is used as a command to 1616/OS, whether the command is typed in at the keyboard, read from a `.shell` file or passed to one of the ***exec*** system calls. This permits the writing of quick transient commands (such as `del`, `ren`, etc) or altering the operation of existing commands (1616/OS searches the memory-resident drivers for a command before it searches the inbuilt command tables). It may also be used to alter the mode of an MRD which performs background functions.

Using buildmrd.xrel

This program permits the linking together of a number of memory resident driver programs, whilst specifying certain system information.

The usage of the program is:

```
buildmrd -sstacksize -vvideosize -rrdsize -ccolours -lnlastlines -ooutfile
-bncacheblocks -fmaxfiles -dndcentries  mrdfile.mrd mrdfiles.mrd ...
```

The parameters all have sensible defaults, and all are optional, so you need only use those you want to alter. You should discard earlier versions of this program, and use the version supplied with 1616/OS Version 4.2 or later. Brief details of parameters are below.

STACKSIZE

is the amount of space reserved for the system stack in kbytes. If this not specified it defaults to 32 kbytes or 64 kbytes, depending on version. If you are heavily into FORTH, etc., you may need more.

VIDEOSIZE

is the amount of RAM at the top of memory (ending at $80000 in an unexpanded system, or 1, 2, 3 or 4 megabytes higher if you have memory in an expansion board) reserved for video display in kbytes. If not specified this defaults to 32 kbytes (it cannot be made less), normally starting at $78000. If you specify 64 kbytes, for example, it will normally start at $70000.

RDSIZE

is the size of the RAM disk in kbytes, and should be a multiple of 8 kbytes. If not specified this defaults to 200 kbytes. It should not be made less than 24 kbytes. If the `mrdrivers` file cannot be found at boot time, the RAM disk size is set according to the setting of bits 0 and 1 of the hardware diagnostic switches.

COLOURS

Colour settings were added to the header structure of the MRDRIVERS file, as at 1989. Bit allocation in this longword is as follows:

00-03	Pallette entry 0
04-07	Pallette entry 1
08-11	Pallette entry 2
12-15	Pallette entry 3
16-19	Border colour

This longword is set using the `-c` flag in `buildmrd.xrel`, when creating the MRDRIVERS file. For example, `-c1fa50` will give a border colour of 1, and pallette entries of 15, 10, 5 and zero (which are actually the default in any case). If the MRDRIVERS file contains zero for the colour setting, the system assumes it is an old version MRDRIVERS file, and sets the colours to the default.

NLASTLINES

Last line recall depth. Set the number of previously input lines retained by the keyboard line editor. Default is 10 lines of history.

OUTFILE

is the output file pathname. It defaults to `mrdrivers`.

NCACHEBLOCKS

Number of blocks of memory retained for disk cache (buffers disk access). These cache system data (root blocks, etc) only, and is not actually used by this version of the operating system.

MAXFILES

The maximum number of file control blocks available to the system.

NDCENTRIES

Number of directory cache entries, for caching directory contents of your path.

MRDFILES

is a list of zero or more memory resident driver files. By convention these filenames end in '`.mrd`'. If no name extension is given, a `.mrd` is automatically added. These programs must comply with the guideslines set out here for memory resident drivers.

Memory resident driver conventions

A memory resident driver program must comply to the system described here if it is to work properly (or at all).

An MRD file is a standard `.xrel` file whose extent has been changed (by convention) to `.mrd`.

The system clears the MRD's **bss** area when it is first loaded at level 0 reset time. From then on the MRD's **bss** area is safe and private. Early releases 1616 C compiler's run-time startup code cleared the bss area on entry to the code. This results in the MRD's bss being zeroed each time the system calls the MRD! The run-time startup code (file `crtapp.s`) must be altered if MRDs are to be written in C.

The main entry point of an MRD is its start address. This address is called whenever the system or a user program performs a ***callmrd*** system call to communicate with the MRD. At this entry point the MRD is passed two numbers on the stack. They are a command number at 4(sp) and a single argument at 8(sp). These are the `cmd` and `arg` arguments to the ***callmrd*** system call. The currently defined commands are listed below. These are a guideline: some of them may not be appropriate for a particular driver. If you are writing a driver with its own commands, do not use commands 0 to 255; these are reserved.

Unless the MRD command is expected to return a value (such as MRD_VERS, etc) it should return 0 in d0.

The MRD code should nominally preserve all registers except d0, d1, a0 and a1, but it is probably safer to preverve all registers without exception.

0	MRD_0RES	Level 0 reset: initialise
1	MRD_1RES	Level 1 reset: initialise
2	MRD_2RES	Level 2 reset: initialise
3	MRD_NAME	Return pointer to driver name
4	MRD_VERS	Return driver version number
5	MRD_ENABLE	Enable driver
6	MRD_DISABLE	Disable driver
7	MRD_DOIT	Do whatever the driver does
8	MRD_STOPIT	Stop doing whatever the driver does
9	MRD_EXECENTRY	Return exec command entry point

Commands 0 - 2
The MRD is called with these command numbers when the system is reset. The MRD should perform any appropriate initialisation, vertical sync vector installation, memory allocation, etc at this time. Return 0 in register d0.

Command 3 MRD_NAME
The MRD must return a pointer to a null terminated string which is the name of the MRD.

Command 4 MRD_VERS
The MRD must return a byte in register d0 which indicates its version number. Bits 0-3 indicate the minor revision level, bits 4-7 indicate the major revision level. For example, $34 indicates version 3.4.

Command 5 MRD_ENABLE
The MRD enables itself. See below.

Command 6 MRD_DISABLE
On receipt of this command the MRD disables itself, so if it is called while disabled it does nothing. If the MRD is an executable command it should set a flag so that it returns zero in response to a subsequent MRD_EXECENTRY command. An MRD should put itself in the enabled mode after any level of reset.

If, for example, the MRD's function is to print the current time on the screen it should not do this when disabled. See the `trd.mrd` for example code.

Command 7 MRD_DOIT
This is the general way of making the MRD do whatever it does. If, for example, the MRD's function is to do a dump to printer of the current screen contents, a call to the MRD with this command should

initiate the printout. If the MRD is currently disabled (via a MRD_DISABLE) then the MRD_DOIT command should be ignored.

Command 8 MRD_STOPIT
This is the way to prevent the MRD from doing whatever it does. In the screen dump code above, for example, an MRD_STOPIT command may abort the screen dump. If the MRD is a print spooler, MRD_STOPIT and MRD_DOIT may stop and start the spooler output.

Command 9 MRD_EXECENTRY
When the MRD receives this command it should return the address of its entry point when called as a transient program. If the MRD does not have a transient command calling mode, return zero for this command.

If the MRD is to be used as a transient program, the address returned here is that of the start of the transient command handler.

The transient command handling code works in just the same manner as a normal transient command `.exec` or `.xrel` program. When the command's name (as returned by the MRD_NAME command) is encountered by the command interpreter, the system stacks the nargs, argstr, argtype and argval values and jumps to the code. The program may open files, print characters, allocate memory, etc. I/O redirection works normally. Error codes may be returned to the system. The ***exit*** system call works correctly.

One difference between transient command MRDs and normal transient programs is that of recursion: rerunning a normal transient from within itself causes a second copy of the program code to be loaded into memory; this does not happen with MRD transients: they are simply reentered when they ***exec*** themselves.

Writing memory resident drivers

The business part of an MRD which is used as a transient program is practically identical to a normal 1616 program - they in fact have little advantage over, say, a `.xrel` file in a RAM disk directory which lies in the head of your execution path, set by the `xpath` command. One significant difference is that transient MRD programs are looked for before the system scans the internal command list; this enables effective modification (by replacement) of 1616/OS commands.

The system uses the MRD_EXECENTRY and MRD_NAME commands every time a command is processed, so transient MRDs may change their name and/or disable themselves at any time.

The major application for MRDs - that for which they were designed - is as pop-up utilities, system call alterations, background task handling, etc. Obviously, an MRD should be re-entrant, as the user in a multitasking system may invoke it more than once.

An MRD may hook itself into the system using one or more of several methods. A number of these intercept the system's functions at a fairly low level: when in doubt, save all registers on entry and restore them before exit!

Background processing at vertical sync interrupt time

The driver must install itself (using the ***set_vsvec*** system call) whenever it receives command 0, 1 or 2. An internal routine is called at the rate specified in the call to ***set_vsvec*** and performs the required processing. If you want it called at more than 50 Hz, use a different interrupt.

Background processing initiated by other interrupts

If the MRD has a routine which is to be called when an interrupt from a device such as a serial channel occurs, it must install a vector to the internal routine in the appropriate vector location (possibly a 'simulated interrupt vector').

A block driver

If the MRD is a block device driver it must install its driver routines with the ***inst_bdvr*** system call at level 0 reset time, when it receives an MRD_0RES command.

Character driver

If the MRD is a character device driver, it must install its driver routines with the ***add_opdvr*** and/or ***add_ipdvr*** system calls at level 0 reset time, when it receives an MRD_0RES command.

Keyboard intercept

The MRD may inspect the passing stream of keystrokes by installing itself in the keyboard scan code processing queue. This is done by calling the ***set_kvec*** system call with a pointer to the processing function whenever a level 0, 1 or 2 reset occurs, and saving the return value in local storage.

The keystroke processing function is called whenever a key is pressed or released. The scan code to be processed is at 4(sp). When the processor has completed its task it should jump to the keyboard handler which was installed when this driver installed itself. The pointer to the old keyscan processing function was returned from the ***set_kvec*** call. Before passing control off to the old handler with a JSR instruction, the key scan code should be pushed onto the stack, so the called code has it available at 4(sp); alternatively the stack pointer may be restored, and control is passed with an indirect JMP instruction.

System call replacement/intercept

The MRD places pointers to its internal routines in the system call jump table by passing them in the ***set_stvec*** system call.

The replacement system call handling call code is called whenever the system call which it services is called (obviously). The call's arguments appear at 4(sp), 8(sp), etc. The handler may pass the arguments to the old system call handler if desired - the ***set_stvec*** call returns the old handler entry point. The stack frame must be identical.

The mrdrivers file

The memory resident driver boot file MRDRIVERS has the following structure: (most values are longwords, well, actually ints)

Offset	Name	Usage
0	magic1	Always $601A Actually a BRA 26 (why?)
4	vers	MRDRIVERS format version (initially 2)
8	rdsize	Size of RAM disk in kbytes
12	memusage	Total of all text, data, bss in all drivers in file
16	ndrivers	Number of drivers in the file
20	magic2	Always $D80AB7F1
24	stackspace	Size of system stack space in bytes
28	vramspace	Space reserved for video RAM (at top of memory) in bytes
32†	obramstart	Address of the start of on-board memory.
36†	mrdstart	Address of the start of the MR drivers
40†	rdstart	Address of the start of the RAM disk
44	vramstart	Address of video RAM start
48†	colours	Pallette and border colours
52	nlastlines	last line recall depth (short)
54	ncacheblocks	disk cache blocks (short)
56	maxfiles	file control blocks (ushort)
58	ndcentries	directory cache entries (ushort)
60	chardrivers	pointer to 16 chardriver structs
64	padd	28 shorts, for future expansion

From here, all the MR drivers, in `.xrel` form are concatenated, including their headers.

The fields marked † above are not initialised in the `mrdrivers` file. They are only written in the RAM copy of the above structure.

3
Character Device Drivers

This chapter covers installing and locating input and output character device drivers.

It also briefly describes the rather elaborate character device structure that provides considerable control over the detailed operation of the I/O ports by means of the ***cdmisc*** syscall. A (somewhat easier) control is obtainable by using the `chdev` program included on the *Users Disk*. This allows alteration of almost all aspects of the character oriented devices, including reactions to signals, end of file characters, xon, xoff, raw mode, hardware handshake, RS232C signal levels, buffer sizes, and much more.

Character device MRDs

Character device drivers may be installed in the memory resident drivers file MRDRIVERS. When the memory resident driver is called with the MRD_0RES command (at level 0 reset time) it installs the character device driver using the ***add_ipdvr*** and/or ***add_opdvr*** system calls, documented below.

The driver installation system calls return the device handle (if there were less than 16 character device drivers currently installed). If the driver is to become standard output, standard error or standard input, then a call to ***set_sop***, ***set_ser*** or ***set_sip*** must be performed every time an MRD_0RES, MRD_1RES or MRD_2RES command is sent to the MRD. Pass the device driver handle to the system call.

Install input driver - add_ipdvr

add_ipdvr(ivec, statvec, name, passval)

d0	10	
d1	ivec	Pointer to character input code
d2	statvec	Pointer to character input status code
a0	name	Pointer to colon-null terminated device identifier name
a1	passval	Value passed to input code & status code
Return	Character driver number or -1 if no room	

An input character device driver consists of a status routine and an input routine. The status routine returns in d0 a non-zero number if one or more characters are available on the input device. The input routine waits until a character is present, then returns it in d0. Negative error codes may be returned.

ivec is the address of the input routine. The input routine is called by the system whenever a read from the character device is required. passval is passed at 4(sp). This routine must wait until a character is available, and then return it in d0. All other registers should be preserved.

statvec is the address of the input status routine. It is called by the system whenever the status of the device is to be determined. passval is passed at 4(sp). Return the status (1 = ready, 0 = not ready) in d0. All other registers should be preserved.

name is the address of a null-terminated string which identifies the driver. The name may be up to 16 characters in length, including the trailing colon and null. The name of the driver must end in a colon for the driver to be recognised as a character device. The name is distinct from the name returned by a memory resident driver when it recevies an MRD_NAME command.

passval is a number which is available for passing to the driver, whenever the input or status routines are called. This permits the installation of multiple character device drivers which in fact all call the same input and status routines. They are installed with the same addresses for ivec and statvec, but with different names and passvals. The ivec and statvec routines then use the passval field to determine which of a range of character input sources is being accessed.

This system call returns the character device driver number. This is the same number as that which is returned when the device is opened with the open system call, or when it is located using the ***find_driver*** system call.

Miscellaneous entry point - add_xipdvr

Character device drivers now support an optional miscellaneous entry point. It is installed using the ***add_ipdvr()*** system call. The normal form of the system call is
add_ipdvr(iovec, statvec, name, passval)
The extended form is
add_xipdvr(iovec, statvec, name, passval, miscvec)
Where the iovec must have bit 31 set to indicate to the system call that miscvec is valid.

add_xipdvr(ivec, statvec, name, passval, miscvec)

d0	10	
d1	ivec	Pointer to character input code **with** bit 31 set.
d2	statvec	Pointer to character input status code
a0	name	Pointer to colon-null terminated device identifier name
a1	passval	Value passed to input code & status code
a2	miscvec	cdmisc handler

Return	Character driver number or -1 if no room

Every input driver MUST have a corresponding output vector, to keep the device handles in the correct order. The input driver must be installed and removed before the corresponding output driver.

Calling ***add_ipdvr()*** or ***add_xipdvr()*** with iovec set to zero will result in the removal of the character device driver.

The miscellaneous vector points to a routine within the driver which is called whenever a program performs a ***cdmisc()*** system call.

The writing of character device drivers is not covered here, but basically the miscellaneous routine receives the following arguments:

4(sp)	The character device driver number (its handle)
8(sp)	The device's passval, as given when its input driver was installed.
12(sp)	The ***cdmisc()*** command (see below)
16(sp)	Argument 1
20(sp)	Argument 2
24(sp)	Argument 3

The driver should return zero for any command which it does not recognise.

Miscellaneous entry points are optional. In 1616/OS only SA: and SB: implement it fully. The CON: device has a miscellaneous entry point simply for the purposes of returning the address of the video driver multi character write routine.

Install output driver - add_opdvr

add_opdvr(ovec, statvec, name, passval)

d0	12	
d1	ovec	Pointer to character output code
d2	statvec	Pointer to character output status code
a0	name	Pointer to colon-null terminated device identifier name
a1	passval	Value passed to output code & status code
Return	Character driver number or -1 if no room	

An output character device driver consists of a status routine, and an output routine. The status routine returns a non-zero number in d0, if one or more characters may be sent to the device (via the output routine). The output routine waits until a character may be sent, sends it and then returns. Negative error codes may be returned.

ovec is the address of the output routine. It is called by the system whenever a write to the character device is required. The character to be transmitted is passed at 4(sp); passval is passed at 8(sp). This routine must wait until the character can be sent, send it and the return. All other registers should be preserved.

statvec is the address of the output status routine. The status routine returns a non-zero number in d0, if one or more characters may be sent to the device (via the output routine) without causing indefinite waits; that is, there is still room in the device driver output buffer or the device is ready to receive another character. Either way, a non-zero return from the status routine indicates that a character may be sent without causing a wait of unknown duration. The statvec routine is called by the system whenever the status of the device is to be determined. passval is passed at 4(sp). Return the status (1 = ready, 0 = not ready) in d0.

name is the address of a null-terminated string which identifies the driver. The string may be up to 16 characters in length, including the trailing colon and null. The string must end in a colon for the driver to be recognised as a character device. The string is distinct from the name returned by a memory resident driver when it receives an MRD_NAME command.

passval is a number which is available for passing to the driver, whenever the output or status routines are called.

Character Device Drivers

Prior to Version 4.2d, each character device driver (input or output) which is installed in the system was identified by the following data structure:

Offset	Name	Size	Usage
0	doio	int	Pointer to input or output code
4	status	int	Pointer to status code
8	passval	int	Value passed to driver at call time
12	name	16 char	Colon and null-terminated name

These are simply a copy of the values passed to the ***add_ipdvr*** or ***add_opdvr*** system call when the driver was installed.

At Version 4.2d, a much more elaborate character device structure was devised. This new structure allows increased facilities for multiple users, and for customising the interface to each device. Programs that write directly to the char device driver tables will still work if they are patching the output driver, but not the input driver.

Offset	Name	Size	Usage
0	doio	int	Pointer to input or output code
4	status	int	Pointer to status code
8	passval	int	Value passed to driver at call time
12	name	16 char	Colon and null-terminated name
28	lastlines	char	Last line recall buffers
32	doip	int	Pointer to input routine
36	ipstatus	int	Pointer to input status routine
40	ippassval	int	Optional number to pass to I/P drive
44	miscvec	int	Pointer to misc entry point
48	sigintchar	int	If received, send signal down
52	eofchar	int	End of File character
56	xoffchar	int	Xoff character
60	xonchar	int	Xon character
64	resetchar	int	Reset the system
68	rxcount	uint	No of chars that have come in
72	txcount	uint	Number of char gone out
76	intsig	ushort	If set,interrupt signal pending
78	hupsig	ushort	If set, hangup signal pending
80	xoffed	ushort	If set, awaiting Xon, O/P driver sleeps
82	kiluser	ushort	If set ***killuser*** call pending
84	hupmode	ushort	DCD loss mode
86	rawmode	ushort	If set, ignore all special character
88	miorvec	int	Multi character read vector
92	miowvec	int	Multi character write vector
96	modebits	ushort	Application specific
98	statbits	ushort	Permission bits

The modebits field is intended for specific applications. The following bits are defined:

0 (1)	cdmb_xlateesc	Escape code translation
1 (2)	cdmb_rxsigpurge	Purge Rx buffer on signal recognition
2 (4)	cdmb_txsigpurge	Purge Tx buffer on signal recognition

Normally you would use the ***cdmisc*** system call (below) to manipulate this table, or if working from the command line, use the `chdev` program (on the Version 4.2 User Disk) to set the values required.

Vary buffer size for a character device - new_cbuf

new_cbuf(dev, addr, len)

d0	81	
d1	dev	Device identifier
d2	addr	Address of new buffer
a0	len	Length of new buffer
Return	0 (-1 if bad argument)	

This system call may be used to install larger circular buffers for the interrupt driven device drivers in 1616/OS. At power-on, the buffer sizes are in the 200 byte region, which is not great for print spooling, etc.

To obtain larger buffer areas, pass this system call a pointer to some free memory (addr), the length of the free memory area (len) and an identifier which selects the device for which you desire more buffering.

The dev argument selects the device:

dev = 0	Replace serial channel A receive buffer
dev = 1	Replace serial channel A transmit buffer
dev = 2	Replace serial channel B receive buffer
dev = 3	Replace serial channel B transmit buffer
dev = 4	Replace parallel printer output buffer
dev = 5	Replace keyboard input buffer

Do not pass a buffer length of less than 64 bytes.

If the buffer is in allocated memory and is to remain in place after the current program has exited, the buffer memory should be obtained from the system using mode 1 for the ***getmem*** system call.

Performing this system call with the addr field equal to zero will result in the standard buffer being restored. The buffer areas are within 1616/OS's data areas. Do this before returning to 1616/OS if the buffers are only temporary.

Get pointer to character device driver table - get_dvrlist

get_dvrlist(ioro)

d0	96	
d1	ioro	Flag: 1 = output drivers, 0 = input drivers
Return	Pointer to character device driver table	

The system keeps two arrays of sixteen of the above structures. One array is for the output drivers, the other for input drivers. An unused entry in the array has a value of zero in the *doio* field.

The ***get_dvrlist*** system call returns a pointer to the start of one of the two arrays. If ioro is zero, a pointer to the output driver array is returned. If ioro is non-zero, a pointer to the input driver list is returned

The find_driver system call - find_driver

find_driver(ioro, name)

d0	95	
d1	ioro	Unused
d2	name	Pointer to device name or a device handle
Return		Device handle or pointer to chardriver structure or error code.

This system call has been worked to provide individual access to each device's chardriver structure.

If name is less than 16 this system call returns a pointer to the chardriver structure for the corresponding character device driver. Otherwise name is assumed to be a pointer to a string such as "CON:" and a search is performed for that character device driver.

If found its handle is returned, otherwise a negative error code is returned.

The cdmisc system call - cdmisc

cdmisc(dvrnum, cmd, arg1, arg2, arg3)

d0	133	
d1	dvrnum	Character device driver number (device handle)
d2	cmd	Command
a0	arg1	Argument 1
a1	arg2	Argument 2
a2	arg3	Argument 3
Return		Varies. Negative if dvrnum is bad.

A large data structure (`chario.h` struct chardriver) is associated with each character device driver. The ***cdmisc()*** system call provides access to this data structure, and to the driver's miscellaneous entry point if it has one. You can make easy use of this syscall from the command line by using the `chdev` program available on the V4.2 *Users Disk.*

`dvrnum` is the handle of the character device. It is the device's file descriptor. It is the number which was returned when the device's driver was installed.

This call will return zero if `cmd` does not require a return value, or if `cmd` requests a call to the device's miscellaneous entry point and it does not have one.

`cmd` defines the mode of the system call. At this stage commands 0 to 31 are acted upon by the ***cdmisc()*** code in the operating system and require no action by the driver itself. This limit of 0 to 31 may present problems at some future time. Other commands are device specific and are acted upon by the miscellaneous code within the device driver.

The values for `cmd` are defined in `chario.h`.

0: cmd = CDM_OPEN
An ***open()*** or ***creat()*** system call upon this device has been performed.

1: cmd = CDM_CLOSE
The device has been closed. Closes and opens do not balance correctly if the user program does not explicitly do it. Character devices are not closed if they were opened by a command line I/O redirection.

2: cmd = CDM_SETSIGCHAR
The device driver compares incoming characters with the 'sigintchar' element in the chardriver structure. When a match occurs a SIGINT is sent to the process which is blocking the shell running from that character device. The 'sigintchar' for CON: is normally $83, (ALT-^C). This call sets the 'sigintchar' to 'arg1'. Set it to 256 ($100) to disable.

3: cmd = CDM_READSIGCHAR
Returns the current setting of the addressed device's sigint char.

4: cmd = CDM_SETEOFCHAR
Sets the end-of-file character for the addressed character device. Set to 256 to disable.

5: cmd = CDM_READEOFCHAR
Returns the current end-of-file char for the addressed device.

8: cmd = CDM_SETXOFFCHAR
`arg1` sets the character which the device driver uses for flow control. This is the 'xoffchar' in the chardriver structure. The driver compares incoming characters with this character. When a match occurs output from the device is suspended until an 'xonchar' is read. The normal 'xonchar' for CON: is $D3, which corresponds to ALT-S. Set to 256 ($100) to disable.

9: cmd = CDM_READXOFFCHAR
Returns the current 'xoffchar' for the addressed device.

10: cmd = CDM_SETXONCHAR
arg1 sets the ‘xonchar’, which, when received, restarts suspended output. Normally $D1 for the CON: driver (ALT-Q). Set this to 256 ($100) to disable.

11: cmd = CDM_READXONCHAR
Returns current ‘xonchar’ for the addressed device.

12: cmd = CDM_SETRESETCHAR
The character device driver compares incoming characters with the ‘resetchar’ field in the chardriver structure. If a match is found the driver performs a ***warmboot()*** system call. This field is $92 for the CON: driver (ALT-^R). This command moves arg1 to the ‘resetchar’ for the addressed device. Use 256 to disable.

13: cmd = CDM_READRESETCHAR
Returns the current ‘resetchar’ for the addressed device.

14: cmd = CDM_SENDSIGINT
This command is performed by the character device driver at interrupt time when it matches an incoming character with the ‘sigintchar’ field in the chardriver structure. The operating system records the character device driver number and will shortly send a SIGINT to the process which is blocking the shell running off that device. This is how ALT-^C works.

15: cmd = CDM_SENDSIGHUP
Like command 14, except a SIGHUP is sent.

16: cmd = CDM_KILLUSER
Like the above, but a ***killuser()*** system call is performed upon the appropriate shell process.

17: cmd = CDM_SETRAWMODE
Sets the ‘rawmode’ field of the addressed device’s chardriver structure to ‘arg1’. If ‘rawmode’ is set all input processing is disabled: SIGINT, resets, xon, xoff characters are all passed through. This facility is provided so that the device may be put in raw mode without having to individually record and disable every magic character in the structure.

18: cmd = CDM_READRAWMODE
Returns the ‘rawmode’ field from the addressed device’s chardriver structure.

19: cmd = CDM_MIORVEC
If the character device driver supports multichar reads it must return the address of its multichar read routine when it receives this command. If the driver does not support multichar reads, it returns 0.

The multichar read entry point is called with the following arguments:

4(sp): Device handle (or file descriptor)
8(sp): Memory transfer address
12(sp):Number of bytes to transfer
16(sp):Pass value with which the input device driver was installed.

The driver returns the number of bytes actually read to the passed address, which must be less than or equal to the passed byte count. The driver should return if an incoming character matches the chardriver end of file character and the device is not in raw mode.

It should return BEC_RPASTEOF error code if an eofchar is read as the very first byte. It should return on a newline character. It should compare each character with the resetchar, eofchar, sigintchar, etc and take the appropriate action, provided the device is not in raw mode.

The multichar read entry point is called directly from within the operating system, so it should preserve the machine registers (except for d0).

The device driver obtains a pointer to the chardriver structure for the addressed device using the ***find_driver()*** system call. For performance these tables should be found at installation time and their addresses should be saved within the driver's storage.

20: cmd = CDM_MIOWVEC
Similar to CDM_MIORVEC, the driver returns its multi char write entry point or 0 if not implemented. The write code is called in a similar manner to the read code, expect that it must only return after all the characters have been sent and it need do no special character processing, except for looking at the 'xoffed' field in the chardriver structure to see if output is suspended. The value passed to the multichar write code at 16(sp) is the pass value which was supplied when the output device was installed, rather than the input device.

21: cmd = CDM_SETHUPMODE
Sets the 'hupmode' field of the device's chardriver structure to `arg1`. The 'hupmode' field tells the character device driver what to do when a loss of carrier (DCD signal) is detected at interrupt time.

hupmode = 1: Send a SIGHUP to all processes which are running from a shell running off the device

hupmode = 2: Perform a ***killuser()*** system call upon the shell running on the device.

hupmode = 3: Send a SIGINT to the process which is blocking the shell running off that device.

It is the character device driver's responsibility to perform these actions at interrupt time when loss of carrier is detected, based on its 'hupmode' field.

22: cmd = CDM_READHUPMODE
Return the addressed device's 'hupmode'.

23: cmd = CDM_HASMISCVEC
Returns true if the addressed device has a miscellaneous entry point vector installed.

24: cmd = CDM_SETUSERBITS
There is a ‘userbits’ field in the char device structure the use of which is basically not defined at this stage, except for bit 0 which, if set, is intended to tell the driver to perform TVI950 escape code to some other escape code translation. This description seems a little dubious. Ask me about it when I work out what it means.

If arg1 is zero, arg2 is ORed into the ‘userbits’.
If arg1 is 1, arg2 is inverted and ANDed into the ‘userbits’.
Otherwise the ‘userbits’ field is returned unchanged.

The following call modes are not handled by the operating system. Any action or return value is handled by the device specific driver code, which may be in the ROMs, or in an MRD.

32: cmd = CDM_SETMODE
arg1 is treated as a pointer to the standard serial I/O programming structure as described in the documentation for the ***prog_sio()*** system call. The system call
cdmisc(handle, CDM_SETMODE, pointer, 0, 0)
is a now a preferable way of programming a serial device, as it should work with other hardware, if added. From the command line, use the `chdev` program for this.

33: cmd = CDM_READMODE
The driver moves the standard serial I/O programming structure to memory pointed to by arg1.

34: cmd = CDM_SETDTR
If arg1 is non-zero, the driver asserts the device’s DTR signal; otherwise it is cleared.
eg: ***cdmisc***(OPEN("SA:", 0), CDM_SETDTR, 1, 0, 0) will assert DTR on serial channel A.

35: cmd = CDM_SETRTS
Same as above, for RTS signal.

36: cmd = CDM_READDCD
Returns non-zero if the addressed device’s DCD signal is currently asserted.

37: cmd = CDM_READCTS
Returns non-zero if the addressed device’s CTS signal is currently asserted.

38: cmd = CDM_READBREAK
Returns non-zero if the addressed device is receiving a BREAK condition. But is this only active when polling, or after any break is received?

39: cmd = CDM_SETHFC
If arg1 is non-zero, sets the device into hardware flow control mode. For the SCC this is the default. DCD qualifies receive data, CTS is used for hardware flow control, DTR is always asserted, RTS is negated when the device receive buffer is nearly full.

When hardware flow control is disabled the SCC asserts both RTS and DTR and then runs in 3 wire mode, competely ignoring the handshake signals. The SCC channel is taken out of ‘auto-enables’ mode.

40: cmd = CDM_SETBREAK
If arg1 is non-zero, start a break condition on the transmitter of the addressed device. Otherwise clear the break condition. Remember to clear this after use, and check the timing required by the other device when using break.

41: cmd = CDM_TXCOUNT
Returns the number of characters still buffered for transmission from the addressed device.

42: cmd = CDM_RXCOUNT
Returns the number of characters which are available in the software receive buffer which the driver maintains for the device.

43: cmd = CDM_TXROOM
Returns the number of characters which can be sent to the device's output channel (via the ***write*** system call, etc) before the write will block due to the software output buffer filling up.

44: cmd = CDM_RXROOM
Returns the number of characters which can still be received on this device before its software receive buffering overruns.

45: cmd = CDM_TXFLUSH
Wait until all buffered transmit characters have been sent.

46: cmd = CDM_TXPURGE
Zero the transmit buffer pointers, dumping all pending output.

47: cmd = CDM_RXPURGE
Zero the receive buffer pointers, dumping all pending input.

48: cmd = CDM_RXPEEK
Returns the next character which will be read from the input device (via a ***read()***, ***getchar()*** system call, etc). If no character is available, returns -1.

49: cmd = CDM_SETTXBSIZE
Sets the size of the device's transmit buffer to arg1

50: cmd = CDM_SETRXBSIZE
Sets the size of the device's receive buffer to arg1

51: cmd = CDM_READTXBSIZE
Returns the current size of the device's transmit buffer.

52: cmd = CDM_READRXBSIZE
Returns the current size of the device's receive buffer.

53: cmd = CDM_VERSION
Returns the low-level device driver version number. For the drivers within 1616/OS (SA: and SB:) the operating system version is returned here.

54: cmd = CMD_READHFC
Returns the hardware flow control state.

4 Block Device Drivers

A block device driver is a collection of routines which handle the transferring of 1024 byte data blocks to and from a physical device. 1616/OS contains five block device drivers: /RD, /F0, /F1, /H0 and /H1. Except for the /RD, these all involving passing the data to the Z80 disk controller card. Full details of the disk drive controller card low level routines are provided in the *Disk Co-processor Card Manual* (SSDCC).

The memory card can also include a SCSI port. At present, this is operated via an additional driver, installed via an MRD. This provides drives /S0, /S1, etc. These SCSI drives may be the same drive used by /H0, H1 etc., however as the memory board SCSI chip is accessed directly by the 68000, /S0 and so on operate considerably faster than /H0.

In future revisions, it is likely that the /S0 drivers will be added to the ROMs.

A total of sixteen drivers may be installed; these will typically reside in the memory resident drivers file, MRDRIVERS.

This chapter also describes calls for locating a block device driver, and a miscellaneous call. The ***bdmisc*** call allows you to pass commands to a driver. The commands include flushing buffers, determining how many blocks are available on the device, and how many are used or free, write protecting devices, disabling a driver, copying root directory to memory, and other handy functions.

Install a device (Version 3)

The ***inst_bdvr*** system call is used to install the driver routines into the system. The description below covers Version 3 of 1616/OS. See below for more on Version 4 multi-block I/O.

Install a block device driver - inst_bdvr

inst_bdvr(br, bw, misc, name)

d0	100	
d1	br	Block read entry point
d2	bw	Block write entry point
a0	misc	Block driver miscellaneous entry point
a1	name	Pointer to name of block driver
Return	Driver number (negative if installation error)	

The system searches for an empty location within its internal block driver tables. If found, this driver is installed and its driver number (in the range 0 to 7) is returned. If a driver with the same name as that at name is found it is replaced by the new driver.

Once installed, block drivers are permanent: level 1 or 2 resets do not remove them.

Driver 0 is /RD, driver 1 is /F0, driver 2 is /F1, driver 3 is /H0 and driver 4 is /H1.

br is a pointer to the machine language routine which reads a block from the device. When the system wishes to read a block from the device, it calls this routine, with the desired block number at 4(sp), and the address to which it is to be read at 8(sp). The driver must attempt to read the block to the address. If an error is detected, a negative error code must be returned in d0. If the read is successful, return 0. Preserve all other registers.

bw is a pointer to the machine language routine which writes a block onto the device. When the system wishes to write a block to the device it calls this routine, with the desired block number at 4(sp), and the address from which it is to be written at 8(sp). The driver must attempt to write the block. If an error is detected, a negative error code must be returned in d0. If the write is successful, return 0. Preserve all other registers.

The error code returned by the read and write code may be selected from among the normal system error codes - see the file system documentation for details. Typical error return values are -2 (write protected), -5 (I/O error) and -6 (Invalid block requested).

misc is a pointer to the driver's miscellaneous function entry point. This is a routine which is called when the system needs to communicate various information with the driver. The miscellaneous routine is called with a command code at 4(sp) and an argument at 8(sp). The currently defined command codes are listed below. If an unknown command is received, ignore it.

Code 1	MB_FLUSH The system keeps track of how long it is since a block device was last accessed. If the device is removable and is about to be accessed and it has not been accessed for more than 2 seconds (approx) then this code is sent. With removable media, this code should be interpreted as meaning that the media may have been changed. If the device driver does not do block buffering then this code has little use.
Code 2	MB_RESET After the system is reset, all block driver miscellaneous routines are sent this code. The reset level is at 8(sp).
Code 3	MB_VERS This requests that the software version number be returned in d0. The 1616/OS floppy disk driver returns the disk controller's ROM version here.

Code 4	MB_NEWDISK If the media is removable, the system checks the special and the date fields of the root block for changes whenever the disk has not been accessed for two seconds. If a change of media is detected, then this code is sent to the block driver miscellaneous entry point. If the driver performs block buffering it must discard all buffered data upon receipt of this command code.
Code 5	MB_DIRREAD The system is about to read one or more directory blocks: this can be used to implement directory caching in an MRD.
Code 6	MB_NDIRREAD The system is about to read one or more non-directory blocks.

The name argument to the ***inst_bdvr*** system call is a pointer to a null-terminated string which identifies the device and its driver. The name must begin with a slash ("/") character. Put the name in upper case, without spaces.

Multiblock I/O (Version 4)

Earlier versions of the 1616/OS file system called the block device drivers once for every 1k block which was to be read or written. This significantly limits the peak performance which can be obtained, due to all the red tape which needs to be updated between calls.

The file system now uses the new ***multiblkio*** system call for all of its block I/O.

This system call separates the requested reads/writes into runs of sequential blocks, and passes the information on to the block device driver which performs the physical I/O. There is a mechanism in the installation of block device drivers, by which the driver can tell the O/S whether or not it supports multi-block I/O. If the driver does not support multi-block I/O, the ***multiblkio*** system call separates the call into the appropriate calls to the driver's single block I/O entry point. This means that the ***multiblkio*** system call should be called for all I/O needs, but only if the OS version is higher than $3f!

Installing multi-block I/O - inst_bdvr

The ***inst_bdvr*** system call has been compatibly altered so that multi-block capable device drivers can be installed, but old style drivers can still be used.

The usage of ***inst_bdvr*** is

inst_bdvr(br, bw, misc, name, bitmap)

d0	100	
d1	br	Block read entry point
d2	bw	Block write entry point

a0	misc	Block driver miscellaneous entry point
a1	name	Pointer to name of block driver
a2	bitmap	of device
Return	Driver number (negative if installation error)	

see section 4.1 of the *Technical Reference Manual* for additional details on the arguments used here.

If the FIRST character of the string pointed to by name is a control-A (ascii code $01) then the system assumes that a multi-block capable driver is being installed, so it

1. Assumes that the real name of the device starts at name + 12. Assumes that misc points not to the miscellaneous entry point, but to the following data structure:

```
dc.l         misc                    The miscellaneous entry point
dc.l         multirw                 The multi-block I/O entry point
dc.l         version                 Driver version type
ds.l         13                      13 longwords, reserved, zero.
```

Note that the driver must still support the single block read and write entry points, for compatibility with earlier OS versions.

The multirw pointer points to code which is passed the following information:

4(sp)	Flag: 0 = write, non-zero = read
8(sp)	Read/write address
12(sp)	First disk block
16(sp)	Number of disk blocks

The driver must use this data to perform the physical I/O, and then return either zero or a sensible negative error code in register d0.

The multi-block I/O system call - multiblkio

multiblkio(dvr, cmd, addr, blockspec, nblocks)

d0	119	
d1	dvr	Block device number
d2	cmd	Function (see below)
a0	addr	Read/write address
a1	blockspec	Start block number OR pointer to block list
a2	nblocks	Number of blocks to read/write
Return	Negative code on error	

The call has four modes:

Sequential write: cmd = 0:

The blocks are written to disk onto contiguous blocks, starting at the block whose number is in register a1 (blockspec), ending at block blockspec + nblocks - 1.

Sequential read: cmd = 1:

Similar to cmd = 0, nblocks blocks are read from block blockspec, to memory at addr.

Random write: cmd = 2:

In this mode, blockspec points to a list of nblocks 16 bit block numbers. The system call sorts out the sequential runs of blocks in the list and performs the necessary calls to the device driver.

Random read: cmd = 3:

As in mode 2, except data is read, not written.

Other system calls

There are system calls that allow you to locate an MRD driver, and pass miscellaneous commands to it. The low level processing of directory commands is described in the next section.

Locate a block device driver - find_bdvr

find_bdvr(name)

d0	102	
d1	name	Pointer to block device name or -1 or 0 to 4
Return	Varies	

This system call is used to obtain a block device driver's name from its driver number, or to obtain a block device driver's driver number from its name.

If name equals -1 then a pointer to the block device driver structure for the currently logged device is returned.

If name is in the range 0 to 7 then a pointer to the corresponding block device driver structure is returned. A negative error code is returned if no driver is installed under the corresponding number.

If name is greater than 7 then it is assumed to be a pointer to the null-terminated name of a block device, such as "/RD", "/H1", etc. The name may also be a volume name such as "/APPLIX", rather than a physical device name. If the named driver is found then a number in the range 0 - 7 is returned. This is the block device driver's number.

The block device driver structure contains many elements and is not documented by Applix. However the first entry will always be the null-terminated name of the block device.

Call block driver miscellaneous function - bdmisc

bdmisc(bdnum, code, arg1)

d0	117	
d1	bdnum or bdvrnum	Block driver number
d2	code	Type of miscellaneous call
a0	arg1	Additional argument
Return	Result of call	

This system call communicates with a block device driver's miscellaneous entry point. A call is made to the misc entry point of the driver whose number is bdnum.

bdnum (sometimes called bdvrnum) is the number of the block device driver.

code is the command which is to be passed to the driver. It may be one of those listed below, or user defined codes. If defining your own command codes to send to a block driver, do not use codes 0 - 255; these are reserved by Applix. code is passed to the block driver at 4(sp). codes $100 to $104 were defined in Version 4.0b, codes $105 to $107 in Version 4.2d. Andrew says he forgot about reserving them. Those uses that are known are listed below.

code =	$1 Flush the buffers.	mb_flush
code =	$2 Reset has occurred.	mb_reset
code =	$3 Return driver version number.	mb_version
code =	$4 New disk has been detected.	mb_newdisk
code =	$5 About to do directory block I/O.	mb_dirread
code =	$6 About to do non-directory block read.	mb_ndirread
code =	$7 About to do blockmap I/O.	mb_bmbread
code =	$100 returns the total number of blocks on the device.	mb_nblocks
code =	$101 returns the number of blocks currently used on the device.	mb_usedblocks
code =	$102 returns the number of blocks free.	mb_freeblocks

code =	$103	mb_readroot
	moves a copy of the root directory entry (the one in the root block) to memory pointed to by arg1.	
code =	$104	mb_reread
	forces the system to re-read the device's bitmap, even if it is a non-removable disk. This should be done after a program modifies the disk bitmap behind the file system's back.	
code =	$105	mb_disable
	disable the block device driver.	
code =	$106	mb_uid0
	only UID 0 can access the device.	
code =	$107	mb_wrprot
	write protect the device.	

arg1 is an additional argument passed to the driver at 8(sp).

The value returned by the block driver miscellaneous entry point is passed on by this system call.

5
File Systems

This section describes files, directories, block devices and their associated data structures.

Directory entries

A directory entry is a 64 byte data structure which describes a file or a subdirectory. There is a single directory entry in a block device's root block (block 0) which describes the root directory (typically blocks 3 - 9 on a floppy).

The directory entry structure is as follows:

Offset	Size	Name	Usage
0	32 char	file_name	32 byte null-terminated name
32	8 char	date	File creation date.
40	ushort	user_id	User I/D
42	long	load_addr	Load address of `.exec` file
46	long	file_size	Length of file/No of dir blocks
50	ushort	statbits	Status bits
52	ushort	blkmapblk	The block which contains the file's block map block / directory start block
54	ushort	magic	If == V4_2magic (d742, etc), ffblocks OK
56	4 ushort	ffblocks	First four blocks of file

FILE_NAME

is the name of the file/directory which this directory entry describes. It contains no "/" characters. It is in upper case and should contain no control characters. If the first entry in this field is zero then the directory entry is not used. When a file is deleted by the ***unlink*** system call the file's blocks are released and the first character of its filename is copied to the end of the file name (31 bytes in) before the first character is set to zero. This permits files to be un-deleted - the results may not be good if the disk has been written to since the file was deleted. Run `fscheck.xrel` across a disk after un-deleting a file.

DATE

is the date of modification of the file/directory which this directory entry describes. Adding new files to a directory does not affect the date in the directory's directory entry.

USER_ID

is the user identification number, which normally defaults to 0.

LOAD_ADDR

is the address at which `.exec` files are to be loaded and executed. It is set to zero for other types of file.

FILE_SIZE

has a different meaning depending upon whether the directory entry describes a file or a directory. If it describes a file, the FILE_SIZE field contains the length of it in bytes. If the directory entry describes a directory (sub-directory), then the FILE_SIZE field contains the number of 1024 byte blocks which the subdirectory occupies. These blocks are contiguous for all directories.

STATBITS

contains file attribute bits. The following bits are defined:

Bit 0 - A	If set, the file has been backed up somewhere.
Bit 1 - D	If set, this directory entry describes a directory
Bit 2 - L	If set, the file/directory described by this directory entry cannot be modified.
Bit 3 - R	Disable others from reading.
Bit 4 - W	Disable others from writing.
Bit 5 - X	Disable others from executing
Bit 6 - S	Provided for symbolic links
Bit 7 - F	Tells whether ffblocks is valid (file under 4 kbytes, Version 4.2)
Bit 8 - H	File to be hidden from directory listing.
Bit 9 - B	Boring. Not to be backed up.

BLKMAPBLK

field use depends upon whether the directory entry describes a file or a directory. If it describes a file, then the BLKMAPBLK field contains the number of the block which contains a list of the blocks occupied by this file, in the order used. If the directory entry describes a directory (sub-directory), then the BLKMAPBLK field contains the number of the block at which the sub-directory's contents start.

MAGIC

A magic number to inform us whether the following four unsigned values are pointers to blocks. The magic number is d742, with the 42 changing with each version of the operating system. This also lets us tell which version of the operating system produced a file.

FFBLOCKS

Four pointers to the first four blocks of a file on a disk. Speeds up disk access because we don't need to check the BLKMAPBLK to locate short (under 4 kbyte) files.

The structure of block devices

A block device consists of the root block (block 0), the block usage bitmap, the boot block, the root directory and the actual storage area.

The root block

The root block is always at block zero of a device. The root block structure defines the structure of the disk. It contains the following fields:

Offset	Size	Name	Usage
0	word	NBLOCKS	Number of blocks on device
2	word	SSOSVER	Version of 1616/OS under which the device was initialised
4	word	BITMAPSTART	The block at which the device block usage bitmap starts
6	word	DIRSTART	The block where the root directory starts
8	word	NDIRBLOCKS	The number of blocks in the root directory
10	word	REMOVABLE	Non-zero if the media is removable
12	word	BOOTBLOCK	Number of block which contains boot code
14	long	SPECIAL	Randomised number for distinguishing disks
18	64 bytes	ROOTDIR	A directory entry which describes the root directory

In detail:

NBLOCKS

The number of blocks on the device. This is read when the disk is logged. If a disk with a certain number of blocks is inserted into a drive which previously contained a disk which had a different number of blocks, the change is detected.

SSOSVER

This is the 1616/OS version number of the disk. The disk structure has changed slightly in the change from 1616/OS V2.4 (version = $24) to 1616/OS version 3.0 (version = $30).

BITMAPSTART

The number of the block at which the block usage bitmap commences.

DIRSTART

The number of the block at which the root directory commences. This field duplicates the BLKMAPBLK entry in the root directory entry ROOTDIR, but has been kept so that 1616/OS V3.0 disks may be read under 1616/OS V2.4

NDIRBLOCKS

The number of blocks in the root directory. This field duplicates the FILE_SIZE entry in the root directory entry ROOTDIR, but has been kept for 1616/OS V2.4 compatibility.

REMOVABLE

This flag is non-zero if the media is removable and the system has to check for swapped disks. There is a performance benefit when using non-removable media; in fact a floppy may be defined to be non-removable. If this is done the system must be reset when the floppy is replaced.

BOOTBLOCK

Contains the number of the block which is read in to memory at $3c00 when booting from disk. If this field is zero then no boot code exists.

SPECIAL

A randomised number which is (hopefully) unique for every disk. The system inspects this field and the DATE field in the root directory entry ROOTDIR for a change which indicates a disk swap.

ROOTDIR

This is a standard 64 byte directory entry. It describes the root directory of the disk, so bit 1 of the STATBITS field is set. The DATE field in this directory entry represents the date of creation of the volume and is also used for detecting disk swaps.

The block usage bitmap

Every block device has a bitmap associated with it which records which blocks are currently used and which are free to be used.

The bitmap is a sequence of bits. A one means that the corresponding block is used; a zero indicates that it is free. The most significant bit (bit 7) of the first byte in the bitmap corresponds to block 0; bit 6 of byte 0 corresponds to block 1, etc. The bitmap may extend over more than one block: one block contains 8192 bits, so one block of bitmap is needed for every 8 megabytes in the disk.

The BITMAPSTART field in the root block identifies the block at which the bitmap starts. If the bitmap is more than one block long, all the blocks must be contiguous, starting with the block identified by BITMAPSTART. The system uses the NBLOCKS field to calculate how many blocks are contained in the bitmap.

The bitmap usually starts at block 1 of a disk. It occupies however many contiguous blocks are required, based upon the number of blocks on the device.

The boot block

The boot block contains the boot program which the system executes at address $3c00 every time the system is reset.

The BOOTBLOCK field indicates whether or not the disk contains a boot block to be loaded at reset time and, if so, what block it is. Whenever the 1616 is reset (by powering on, pressing the reset switch or by Alt Ctrl R) the operating system performs all initialiation and then goes through all the block drivers in order (/RD first, then /F0 then /F1 then any others) looking for a device with its BOOTBLOCK field in the root block non-zero. When such a device is found the indicated block is loaded into memory and executed at address $3C00 in the 1616.

The current level of the reset (0, 1 or 2) and the block driver number from which the system is booting are passed on the stack at 4(sp) and 8(sp) respectively. This allows the boot code to perform whatever level of initialisation is needed.

Note that the RAM disk may be used in this manner. Block 3 is reserved for booting, however it is not normally used. To use it, read in the root block, set the BOOTBLOCK field to 3, write out the root block again and then put your boot code in block 3. Remember that the 1616/OS OPTION 8 must be set to write enable the system tracks of a block device.

If the disk in drive 0 (/F0) does not contain a boot program, then the system will attempt to boot from disk 1 (/F1). If there is no second disk or if the second drive has no disk in it then the attempt to read from the second disk will fail, taking a few seconds to time out. Look in the SYS directory of the 1616/OS release disk for the file `boot3v.exec` or `bootv3`. This is the standard boot code.

The root directory

The disk root directory is a number of blocks reserved for the home directory of the disk. Its size is set when the disk is initialised. On floppies the `blockdev.xrel` program does this. The root directory should start after the root block, bitmap block and boot block.

Formatting media

There is no facility here for formatting and initialising the media. A dedicated program, such as the floppy disk utility `blockdev.xrel`, must be written to do this for each block device.

If the removable field in the root block is set, the device's root block and bitmap are only ever read once, after a reset. You can really stuff up someone's day by fudging up a removable disk this way.

It is important that all disks have different values in their root block special and date fields; if two disks have the same values the system will not detect disk swaps and may corrupt disks. For this reason disk copying programs MUST alter the special and/or the date fields in the root block when copying 1616/OS disks. The `diskcopy.xrel` utility program does this.

Utility programs

A number of utility programs are supplied on the 1616/OS V3.0 User's disk.

Initialising block devices

The program `blockdev.xrel` permits the initialisation of block devices, formatting of floppies and the conversion of 1616/OS V2.X disks to 1616/OS V3.0

To run this program type

```
blockdev devname
```

Where `devname` is the name of the device (/F0, /F1, etc) which you wish to format, initialise, etc.

There are three levels of disk preparation available with this program. The most basic level is to simply alter the disk's boot sector program; you enter the name of the new boot program and this is written onto the disk. If no boot program exists on the disk then the disk is skipped in the booting sequence.

In the next level of disk preparation you may 'initialise' a disk. This recreates the disk's root block, bitmap and directory. All files are lost. The boot block needs to be rewritten after this.

The next level of disk preparation involves a physical format of the disk, followed by initialisation, followed by the boot code setup. This will only work on the /F0 and /F1 devices, since these are the only physical devices which this program knows how to format.

Another option with this program is to upgrade a disk to 1616/OS V3.0. This alters the disk's root block into the correct format. The V2.X directory becomes the V3.0 root directory. The disk is still readable and writeable under 1616/OS V2.X. This option may be used for altering a disk's name, as well as upgrading the version.

Copying disks

The program `diskcopy.xrel` copies a 1616/OS disk block-for-block to another disk. The DATE and SPECIAL fields in the rootblock of the target disk are altered to prevent the system from assuming the two disks are the same. Uses multiblock I/O, if available.

Usage of this program:

```
    diskcopy sourcedev destdev [-f] [-v] [-r] [-s]
```

Where `sourcedev` is the device identifier of the source disk (/F0, /F1, etc) and `destdev` is the destination disk identifier. If these are the same then you will be prompted to perform disk swaps at the appropriate times.

If a bad block is detected on the source disk during reading then a block of zeroes is written to the destination disk in its place.

The `-f` flag forces physical formatting of the destination disk. This only works if the destination device is /F0 or /F1.

The `-v` flag sets verbose mode: the program prints out more status information as the copy proceeds.

The `-r` flag suppresses the re-randomisation of the root block's DATE and SPECIAL fields, yielding an exact copy of all blocks of the disk.

The `-s` option suppresses the use of multi block I/O, and copies a block at a time.

An example of the use of this program:

```
    diskcopy /F0 /F1 -v -f
```

This will copy from /F0 to /F1, formatting /F1 and printing out status information as the copy proceeds.

Checking and repairing file systems

The `fscheck.xrel` utility scans the file system on a disk, reporting any inconsistencies in information on the disk. The old version can not handle files lager than 512 kbytes.

To use this program, type

```
    fscheck devname [-v] [-y] [-yy] [-q]
```

Where `devname` is /F0, /F1, etc.

The `-v` option, if specified, causes the program to operate in verbose mode, so more status information is printed.

If the `-y` option is provided this program automatically answers ‘yes’ to all its questions, except for the last one where it asks if you wish to write out the disk bitmap; this question must be manually answered.

If the `-yy` option is given all questions, including the last one, are automatically answered in the affirmative.

The `-q` option does a quicker check, that is not as thorough. If a disk passes this, it is correct. If it fails, use the other flags and run again to fix it.

`fscheck.xrel` does not check the disk for bad disk blocks - these may or may not be detected, depending upon where on the disk they lie.

This program performs the following sequence of operations:

1 The file system is descended; information about every file/directory on the disk is read. Checks are made for blocks used by files or directories which are in the range 0 to the end of the root directory, or are beyond the capacity of the disk.

2 The disk bitmap is read

3 For every block on the disk the following error conditions are checked for:

    3a Block in range 0 to end of root directory not reserved

    3b Block common to two or more files/directories

    3c Block reserved, but not in a file/directory

    3d Block used in a file/directory but not reserved

The program can fix all these errors except that in 3b: if two or more files share the same block, you have to manually delete one of them. It is important that `fscheck.xrel` be re-run immediately after this has been done.

When the program encounters an error in 3a, 3c or 3d it will report it and ask whether or not to fix the error. If a ‘Y’ is entered a copy of the disk bitmap is altered. After all blocks have been processed you will be asked if you wish to rewrite the altered bitmap image. At this stage no changes have been made to the disk. Answering ‘Y’ here causes the altered bitmap to be written out.

Errors detected in operation are repaired by deleting the offending block from the file. The file length is reduced by 1024 bytes (unless the block is at the end of the file). A message is displayed and you are asked whether or not the fix is to be made. The file will probably be corrupted. If a file’s block map block number is an invalid block then the file is deleted. If a directory’s block range is invalid then it is removed. The user is told about all of these things and prompted to confirm. A number of these fixes will cause bitmap inconsistencies, but these are fixed later or on another pass of the file system check program.

This program should be rerun until it gives the disk a clean bill of health; it may take several passes to fully repair a disk.

If a change is made to the bitmap of a non-removable disk this program exits with a ***warmboot*** system call, which is like pressing reset. This must be done because the system never re-reads the bitmap and root block of a non-removable device, so alterations to the bitmap would not be noted if this program exited in the normal manner.

Directory Manipulation

A number of low level calls are available to programmers in Version 4. These would normally only be used if higher level calls are unsatisfactory for your purposes.

Manipulate directories - processdir

processdir(pathname, buf, mode)

d0	118	
d1	pathname	Pointer to name of file/directory
d2	buf	Various usage, mainly a pointer
a0	mode	Processing mode
Return	Negative error code or directory position	

This is the internal directory manipulation function. It is a low-level function and should only be used where other system calls are deemed unsuitable. ***processdir*** scans directories and alters or reads their contents.

There are eight modes. In those modes where pathname is actually used as a path it may be a relative path, such as ../dir1/file1, etc.

If an error is detected a negative error code is returned. Otherwise the return value in most modes is the position on the disk of the relevant directory entry. Bits 0-15 of D0 represent the disk block in which the entry resides; bits 16-31 are an index into the block of the directory entry. The index is in the range 0-15 and must be multiplied by 64 (the directory entry size) to obtain the offset into the block. The exception to this is when mode 2 is used to read the directory entry for a root directory (such as /RD). This directory entry resides in the root block, not in a directory, so the directory index return is rather meaningless.

Mode = 0	delete file or directory The file or directory defined by path is unlinked. buf is unused. The directory position is returned.

Mode = 1	rename file The file_name field in the directory entry of the file or directory defined by path is replaced by the name at buf. The directory position is returned.
Mode = 2	read directory entry The directory entry for the file or directory defined by path is read to memory at buf. The directory position is returned.
Mode = 3	write a directory entry The directory entry for the file or directory defined by path is over-written with the directory entry at buf. The directory position is returned.
Mode = 4	not used
Mode = 5	write to empty position The directory which path would live in if it existed is scanned for an empty entry, and if one is found the directory entry at buf is written in. If, for example, path is '/F0/dir1/dir2/myfile' then the /F0/dir1/dir2 directory is scanned. The directory position is returned.
Mode = 6	change date The directory entry for the file or directory defined by path has its date field altered to the data at buf. The directory position is returned.
Mode = 7	clear attribute bits The directory entry for the file or directory defined by path has its status bits field AND'ed with the one's complement (inverse) of buf. Note that buf is not being used as a pointer here. The directory position is returned.
Mode = 8	set attribute bits The directory entry for the file or directory defined by path has its status bits field OR'ed with buf. Note that buf is not being used as a pointer here. The directory position is returned.

Check permissions - chkperm

chkperm(pdirent, mask, path)

d0	141
d1	pdirent
d2	mask
a0	path

Return

Checks that the current user is permitted to access the file or directory described in fullpath. A copy of the file's directory entry is pointed to by pdirent. This routine has been made a syscall so that more extensive permission checking may be done, based on the full pathname.

Symbolic links - slink

slink(one, two, three)

d0	134
d1	one
d2	two
a0	three

Return

Ask Jeremy Fitzhardinge, who wrote it.

6 Other Calls

Many system calls operate at a low level in the OS. Normally the more limited higher level calls should be used, when available.

In particular, expansion of wildcards, and interpretting of command line arguments are supported. Installing and altering system call vectors permits the easy replacement of system calls by your own code.

The mousetrap call is partly documented. The addition of multitasking also requires more access to system flags, such as when the Z80 is busy. Command substitution and additional names for commands is made easier by the ***alias*** call.

There is an extensive general purpose ***oscontrol*** call. This includes a considerable number of modes (36 at the moment). You can selectively disable 1616/OS commands, find the starting address of on-board ram, force a subsequent reset to be Level 0, read the xpath and assign, set and read the file creation mask (umask) or the user number (UID), initialise the video or the keyboard, set or read the beep volume, or its vector so you can use your own sounds. You can find the SCC ISR routine, lock in multiblock I/O to a block device, find out if the system has been powered up, or merely reset, alter the environment strings.

Low Level Calls

In versions before 3.0, the low level calls were not available to external programs, but they have now been made public. Example code fragments are given for these calls.

Process command strings - sliceargs

sliceargs(str, argv, wcexp)

d0	93	
d1	str	Pointer to string to process
d2	argv	Pointer to array of 256 pointers (1024 bytes)
a0	wcexp	Flag - can expand wildcards
Return	Number of arguments or error code	

This is an internal 1616/OS function which could be handy and so has been made public. In particular it gives user programs a relatively simple way of expanding wildcard representations of pathnames into multiple pathnames.

sliceargs takes as input a string consisting of words, separated by whitespace (such as a command typed into 1616/OS). The separate words within the string are peeled off and stored in memory. Space for them is obtained with the ***getmem*** system call, with mode = 0. Longword pointers to the separated words are placed in the argv array. The return value is the number of words separated. A nil-pointer is put in the argv array to indicate the end of the valid arguments.

It is the caller's responsibility to free the memory pointed to by each element in the argv array after use. The string at str is not altered.

If the wcexp flag is non-zero then any non-quoted wildcards are expanded before they are separated. This allows ***sliceargs*** to be used simply for wildcards expansion.

Strings surrounded by quotes are treated as single words.

Example: Suppose that in an application program there is an option to perform some operation upon some files. The names of the files are to be supplied by the user and we wish to permit wildcard expansion.

The steps to do this are:

1 Get the string from the user which contains the filenames

2 Assume the string is pointed to by register a0:

```
            move.l       #93,d0        * sliceargs syscall
            move.l       a0,d1         * pointer to string
                                       * to interpret
            move.l       #argvbuf,d2 * array of pointers
            move.l       #1,a0         * Permit expansion
                                       * of wildcards
            trap         #7            * Do it
            tst.l        d0            * Error?
            bmi          errorhandler  * Complain to user
            <more code>

argvbuf     ds.l         256           * Room for pointers
```

3 The argvbuf array now contains pointers to all the constituent words in our original array. Now use them:

```
            move.l       #argvbuf,a3 * Pointer to pointer
                                       * to current string
loop        tst.l        (a3)
            beq          done          * Nil pointer:
                                       * all processed
            move.l       (a3),a0       * Get pointer
                                       * to string
            move.l       a3,-(sp)
            bsr          dothings      * Process the
                                       * string at (a0)
            move.l       (sp)+,a3
            move.l       a3,d1         * Prepare to
                                       * free the memory
            move.l       #64,d0        * freemem system call
            move.l       a3,-(sp)
            trap         #7
            move.l       (sp)+,a3
            add.l        #4,a3         * Next element in
                                       * the argv array
            bra          loop
```

Interpret and evaluate arguments - clparse

clparse(pargs, ptype, pval)

d0	91	
d1	pargs	Pointer to array of pointers to strings
d2	ptype	Pointer to array of ‘type’ bytes
a0	pval	Pointer to array of evaluated numbers
Return	Nil	

This is an internal function which is used during the processing of commands in ***exec***. It takes as input an array of pointers to strings, which is pointed to by pargs. The last pointer in the array must be a nil pointer (value zero). The strings are null terminated and would not normally contain white space.

ptype is the address of an array of 256 bytes in the user program’s memory areas.

pval is the address of an array of longwords in the user program’s memory areas. The number of longwords required is equal to the number of arguments at pargs. The maximum is 256 longwords.

The strings pointed to by the pargs array are examined to determine whether or not they are numeric. The preceeding ‘.’ for decimal and ‘%’ for binary are understood. The default radix is 16 (hexadecimal).

If it is determined that a string is a valid number, then the corresponding entry in the ptype array is set to 2. The number is evaluated and placed in the corresponding entry in the pval array.

If the entry is non-numeric the corresponding entry in the ptype array is set to 4. A pointer to the actual string is placed in the corresponding entry in the pval array.

A value of 1 in an entry in the ptype array indicates that the corresponding entry in the pargs array did not exist. This may be used to calculate the number of arguments in the pargs array.

Alter/install a system call vector - setstvec

setstvec(vecnum, vector)

d0	80	
d1	vecnum	Number of system call
d2	vector	System call handler entry point
Return	Old vector	

This system call permits the alteration of how a specific system call is handled. Please use sparingly and/or let Applix know what has changed, so that the system calls can remain a useful standard set of commands for all 1616 programmers and users.

When a system call occurs the system stacks a2, a1, a0, d2 and d1, then jumps to the code pointed to by an entry in a RAM jump table, indexed by the system call number. The ***setstvec*** system call permits the alteration of entries in this table. It is passed the number of the system call to vary, and the new entry point for the system call handler. The return value is the old system call handler entry point.

The RAM copy of the system call jump table is reinitialised at all reset levels.

A new system call handler may expect the first argument to the system call (the one which is originally passed in register d1) at 4(sp), the second at 8(sp), etc. The call number is not considered to be an argument. The new system call handler may jump off to the old system call handler after processing the arguments; stack discipline must be maintained: push the required arguments onto the stack in reverse order and JSR to the code which is pointed to by the address returned from the call to ***setstvec***. Remember to unstack the arguments and return an appropriate value in d0 upon completion.

If vector is zero, the default setting of the pointer is written into the system call jump table: this may be used to restore the system's normal system call handlers.

If vector is negative the current vector setting is read from the table but no changes are made.

Mouse, alias and disk lock calls

Mouse operations are not completely documented as yet, and may be subject to extensive alteration. The ***alias*** call allows you to readily provide alternative names for commands. It is best seen by examining the code for `alias.c`.

The addition of multitasking means that the operating system must be re-entrant. Where this is not feasible, such as in the file system, flags must to used to indicate when a resource is already busy.

Install mouse driver intercept - mousetrap

mousetrap(trapno, vector)

d0	46	
d1	trapno	Intercept number
d2	vector	User code entry point
Return	Previous vector	

This system call has been made available for development purposes. A standard mouse driver will be included with the release of Version 3.1.

To maintain the mouse pointer on the screen, the driver needs to know when the system is about to overwrite the area of the screen which holds the pointer, so it can be removed before the display is changed, and replaced afterwards. The ***mousetrap*** system call permits the driver to install vectors which point to user-written routines which the system is to call before and after scrolling the screen and drawing characters on the screen.

There are four vectors maintained within the system:

Vector 0 (MT_PCVEC) points to code which is to be called just before a character is placed on the video access page. The following arguments are passed to the user code:

4(sp)	Row number where character is to be drawn.
8(sp)	Column number where character is to be drawn.
12(sp)	Ascii code of character to be drawn.

Vector 1 (MC_ACVEC) points to code which is to be called just after a character is placed on the video access page.

Vector 2 (MC_PSVEC) tells the driver to replace its cursor.

Vector 3 (MC_ASVEC) tells the driver to remove its cursor.

All of the above screen positions are in terms of 8x8 character cells, on the absolute screen.

These vectors are normally zero, in which case the system does not use them. If any of the vectors are non-zero the system does the indirect jump at the appropriate time. A user program installs a vector by putting its number (0 - 3) in trapno, the pointer to the user code in vector and performing the ***mousetrap*** system call.

The ***mousetrap*** system call returns the previous setting of the vector. The user code should save this and, when the system jumps to the user code to which the new vector points, you should inspect the old vector which was returned at installation time. If it is non-zero, set up the correct stack frame and call the code pointed to by the old vector. This permits daisy-chaining of programs which set these vectors.

The user intercept code should preserve all registers and return with an RTS instruction.

All four vectors are zeroed at all levels of reset.

If bit 31 of trapno is set, the appropriate vector setting is returned, but no change is made.

The pre- and post-character output intercepts (those corresponding with vectors 0 and 1) are hooked into the ***rawvid*** system call. This system call is reentrant, so your intercept code must be reentrant also. This means that if the code is executing and an interrupt occurs, during which the code is called again, the servicing of the interrupt character output should not result in any disturbance to the original code when the interrupt routine returns. The time-date on screen MRD uses ***rawvid*** to place the time and date on the screen.

Extra mousetrap modes are available as trapno 4, 5 and 6. Andrew says these are used by vcon, but it is too hard to document, and he's too embarrassed. These are listed in the syscalls as follows:

savevcontext(4, pointer), which saves the video context.

restvcontext(5, pointer), which restores the video context.

vcontextsize(6), which returns the size of the video context.

The video drivers external definition file includes all the information required about the window in use. Most information is store as unsigned shorts.

mode_640	if true, 640 mode
curs_rate	flash speed
curs_enable	turn cursor on or off
thisfg	foreground colour
thisbg	background colour
cur_vap	the crrent video access page
cur_vdp	the current video display page
vstate	terminal emulation state
esc1save	save for second character in ESC sequence
esc2save	save for third character in ESC sequence
blkcurs	block cursor flag
waswrap	flag for suppressing newline after wrap
insertmode	character insert mode

x_start	(a copy of the current window definitions follow)
y_start	
x_end	
y_end	
bg_col	
fg_col	
curs_x	
curs_y	

ourcurs	hardware cursor follows output cursor
x_size	
vidomode	if true, no esc or control processing
y_size	
curpmode	current graphics plot mode
gfgcol	graphics foreground colour
ogfgcol	users's foreground colour
plotfunc	integer pointer to current graphics dot function
underline	current underline mask (uint)
g_xsize	uint
g_ysize	uint
g_xstart	int
g_ystart	int
cur_window	pointer to current window structure
vidram	ushort pointer o start of video access ram
chtab	ushort pointer
curs_count	ushort flash counter
curs_mask	ushort, the flashing mask (all 8 byte of it)
pad	long

Alias a command - alias

alias(cmd, arg)

d0	135	
d1	cmd	commands
d2	arg	entry point or alias handle
Return		Various or error

The six commands available are:

AL_SETVEC	(0)	Set a vector
AL_READVEC	(1)	Read a vector
AL_RMVEC	(2)	Remove a vector
AL_VECTABLE	(3)	Return the table address
AL_TABLESIZE	(4)	Return the size of the table
AL_RMALL	(5)	Remove all aliases

Alias allows command substitution by returning a pointer to a table of substitute entries. See the `alias.c` program for sample code. The table of up to 100 entries (default) can be logged on disk, in the default file `/usr/lib/aliases`. Definitions may be recursive. The `alias.c` program accepts the parameters ON, OFF, EXIT, LIST and MAX_ALIASES.

Arbitrate Z80 use - z80lock

z80lock(cmd, arg)

d0	127	
d1	cmd	Mode for this call
d2	arg	General use
Return		Various

1616/OS maintains a flag which determines whether or not the Z80 is free. This system call lets you use this flag to arbitrate between different processes which directly communicate with the Z80.

cmd = 0	Returns value of the interlock flag. 0 = free, non-zero means the Z80 is being used.
cmd = 1	Returns a direct pointer to the interlock flag. The flag is a longword. This permits quick access to the flag.
cmd = 2	Returns the PID of the process which currently has control of the Z80. Naturally this is only meaningful when the Z80 is currently being used.
cmd = 3	Returns the address of a longword which is used to save the PID of a process when it gains control of the Z80.
cmd = 4	Gain access to the Z80, with timeout. The system call returns under two conditions. A timeout. In this case, arg specifies the timeout duration in 20 millisecond ‘ticks’. If the Z80 is not available within this time, the call is abandoned, and 1 is returned in register d0. If the Z80 is available, and access is obtained, all other processes are locked out. The Z80 **must be freed** if it is to be used again by any other process. A return value of 0 in register d0 indicates the Z80 is available.
cmd = 5	Waits forever until the Z80 is free, and only then returns with control of it. The Z80 must be freed after use.
cmd = 6	Free the Z80, for use by other processes. If the Z80 is not currently reserved, a negative error code is returned in register d0.

If a process is to use this request/grant system call to directly talk to the Z80, then it **must** install a signal handler for itself. This is to prevent it being killed off while it has access to the Z80. If no signal handler is installed, and the process is killed, the file system will be left in a zombie state.

The simplest signal handler that will do the job is an `rts` instruction, installed thus:

```
move.l       #129,d0                 ; proccntrl system call
move.l       #12,d1                  ; mode 12, sigcatch(vector)
move.l       #return,d2              ; the signal handler vector
trap         #7
rts
;
; signal handler for this process.  We are presented with the
; signal types at 4(sp) and an argument at 8(sp)
;
return:
rts                                  ; stubbed
```

Miscellaneous system alterations - oscontrol

oscontrol(cmd, arg, arg2, arg3)

d0	125	
d1	cmd	Command number
d2	arg	Various usage
a0	arg2	Varies, not used in most modes
a1	arg3	Varies, not used in most modes
Return	Varies	

oscontrol is a general purpose system entry point for doing things which do not deserve a system call of their own. It has expanded somewhat, and now includes 36 modes!

As many modes in this syscall have unrelated functions, the different modes each have their own name and macro definitions in `syscalls.h` and `syscalls.mac`.

cmd = 0 ***setibcvec***(vec)
arg (known as vec) points to a vector of bytes which is used to selectively disable 1616/OS inbuilt commands. The purpose of this is for removing the availability of the more damaging commands (such as `syscall .101`!) for dial-up use.

The vector is a string of bytes, each of which corresponds to an inbuilt command. This is used by the current process's home shell process, and remains in force until it is either changed or removed, or until the last process attached to this process's home shell exits.

If vec is zero, all inbuilt commands are enabled.

When a process sets its home shell's disable vector, commands are disabled for all processes which share that home shell process. If one of these processes invokes a new shell type process, that process inherits a copy of its parent's command disable vector, so that even if the oroiginal shell process exits, the new disable vector remains in force.

The bytes in the vector can have values of 0, 1 or 2. Zero means a command is enabled. One means it is disabled (and is bad karma unless you have a way out. Byte value of two is the most useful. If a user executes a program which changes its UID to zero, then that program can execute any inbuilt command (presumably the reason for it changing its User ID).

The command list and byte positions are set out below. The list is in historic order of which command was developed first.

mdb	0	mwaz	23	edit	46
mdw	1	cio	24	<unused>	47
mdl	2	terma	25	quit	48
mrdb	3	termb	26	tarchive	49
mrdw	4	serial	27	pause	50
mrdl	5	fkey	28	tverify	51
mwb	6	setdate	29	ascii	52
mww	7	date	30	time	53
mwl	8	copy	31	xpath	54
mwa	9	syscall	32	option	55
mfb	10	delete	33	volumes	56
mfw	11	cat	34	touch	57
mfl	12	rename	35	filemode	58
mfa	13	dir	36	type	59
mmove	14	msave	37	<ssasm>	60
mcmp	15	mload	38	assign	61
msearch	16	tsave	39	ps	62
base	17	tload	40	kill	63
expr	18	dirs	41	wait	64
go	19	itload	42	set	65
srec	20	cd	43		
help	21	echo	44		
move	22	mkdir	45		

`cmd` = 1 ***obramstart***()
Returns the address of the start of on-board RAM. Zero if not expanded, $100000 if you have a megabyte of external ram, etc.

`cmd` = 3 ***readibcvec***() also known incorrectly as ***curibcvec***
Returns pointer to the inbuilt command disable vector for the current process's home shell process. A return of zero indicates no commands are disabled. The 65 current commands are taken in the same order as in the Eproms. See Andrew McNamara's `disable.c` for a short example of code using this call, with all commands noted.

`cmd` = 4 ***forcelevel0***() also known incorrectly as ***trashwrupword***
Clears the system's power-up words so the next reset, `syscall(101)`, `syscall(1)` or Alt Ctrl R will result in a level 0 reset, not the normal level 2 reset.

`cmd` = 5 ***readxpath***(arg) also known incorrectly as ***xpname***
`xpath` readback. Returns a pointer to the null-terminated name of execution path number `index` (argument in d2). Returns -1 if `index` is larger than the number of `xpath`s currently set. If no valid `xpath` is given, returns <=0.

`cmd` = 6 ***readassign***(arg) also known incorrectly as ***getassname***
Used to read back the current assignments as set by `assign`. `Index` (argument in d2) is used to index into the assignment arrays. If bit 31 of `index` is clear, a pointer to the old part of the assignment is returned. If bit 31 is set, a pointer to the new part of the assignment is returned.

If bit 30 of index is set, returns the User ID of the assignment. If the index is out of range, 0 or -1 is returned.

cmd = 7 ***setumask***(arg)
sets the current process's file creation mask (umask) to arg, for the current process. Used in setting the mode bits in any files or directories that are created. A process's umask is passed on to any child processes when they are created. The umask bits are defined in the file `files.h`. The default is new files can not be read, written or executed by a different user. Don't set the locked bit, or copy and other things fail. Returns the old umask.

cmd = 8 ***readumask***()
Returns the current process's file creation umask.

cmd = 9 ***setuid***(arg)
Sets the current process's user ID (UID) to arg. This UID is inherited by any child processes.

cmd = 10 ***readuid***()
Returns the current process's UID.

cmd = 11 ***get_bdev***(path)
Extracts block device driver number from a relative pathname. Some system calls need this, so putting a pointer to a pathname in path and performing this call returns the number of the block device driver on which that file exists, or an error code if path is invalid.

cmd = 12 ***dumplastlines***(arg)
Frees all last line information associated with the character device whose handle is arg. Used by the memory manager to defragment and increase free memory when ***getmem*** fails. May be used to prevent successive users from seeing what others have typed. The presence of this command means that a process's last line history can be dumped by another process at any time. If your last lines mysteriously disappear, then it probably means that a process has requested more memory than was available. If arg is -1, dumps **all** last lines for all devices.

cmd = 13 ***setwildcomp***(vec)
Sets wildcard comparison vector. vec is a pointer to a new wildcard comparison function. Allows you to displace the existing wildcard matching in the ***sliceargs***() syscall with a better one, if you happen to have a full regular expression matching code hanging round. If vec is zero, the default comparison function is used.

cmd = 14 ***readwildcomp***()
Returns a pointer to the current wildcard matching function, which may be in ROM or external.

cmd = 15 unused (formerly disk cache manipulation).

cmd = 16 unused (formerly disk cache manipulation).

cmd = 17 ***video_init***(level)
Video reinitialisation. Causes a call to the video init code which the system uses at reset time. level is used in place of the reset level, with zero causing the same initialisation as a level zero reset, etc. Puts the video back into a sane state, if you have been doing a few too many experiments.

cmd = 18 ***kb_init***()
Ditto for the keyboard.

cmd = 19 ***setbeepvol***(vol)
vol becomes the new beep volume. Initial value is 50. All polite sound routines should scale their output by this value. Do not set this greater than 127.

cmd = 20 ***readbeepvol***()
Returns current beep volume. Programs that make sounds should first read this setting, and scale their output levels to comply with the user's desired loudness level.

cmd = 21 ***setbeepvec***(vec)
Installs a new beep vector. If vec is zero it is changed to the default beep code in the ROMs. vec points to a piece of code which receives the following arguments:

4(sp): Non-zero if a different beep sound is desired (caused by esc b sequence)

8(sp): 'Length' is 1000 or 2000. Ignore this.

The new beep code may call FREETONE(), but should not perform file system calls, memory manager calls, etc. Your code should preserve all registers. Any new beep code should scale its output by the current volume setting.

cmd = 22 ***readbeepvec***()
Returns a pointer to the current beep vector function.

cmd = 23 ***nouseffbs***(arg)
If arg is one, the first-four-block file system tweak in the directory entry is disabled on reads. If arg is 0, the FFB information is used to locate files less than 4 kbytes long. This call should not be required, but can be used to prevent confusion between files written under different operating system versions.

cmd = 24 ***readffbs***()
Returns state of the first-four-block disable flag, set above. Normally zero.

cmd = 25 ***setmemfault***(mode)
If mode = 0, the current process will be sent a SIGSEGV (signal 11) on memory allocator failures (default). If mode is 1, the process will not be sent a signal. Other values of mode just read the flag for this process.

cmd = 26 ***rxtxptr***(n)
If n = 0, return pointer to the ROM's SCC (serial) transmit interrupt service routine. Otherwise return pointer to the receive ISR. This is here for borrowing the ROM code for use with any additional SCCs attached to 1616 bus.

cmd = 27   ***setbdlockin***(mask)
mask is a bit mask used for making the operating system perform a process lock-in around the ***multiblkio*** system call for that device. If, for example, bit 0 of arg is set, ***multiblkio*** calls involving /RD will be performed in a locked in state. Can only be used if the device driver does no ***sleep***() or ***snooze***() system calls. This call is only useful with the /s0-/s3 direct SCSI drivers under a heavy process load, such as the MGR window manager, giving better disk throughput.

cmd = 28   ***readbdlockin***()
Returns current block device driver lockin mask.

cmd = 29   ***startofday***()
Returns non-zero if this is the very first level 0 reset since the 1616 was powered up, rather than a ***coldstart***() syscall .101.

cmd = 30   ***pfastchar***()
Returns a pointer to an internal very low-level fast multicharacter write to video routine, called ‘fastchar’. ‘Fastchar’ is called with two arguments on the stack. One is a pointer to a string of characters, the second is a byte count.

```
; from assembler
            move.l        #125,d0
            move.l        #30,d2
            trap #7                      ; get pointer to fastchar
            move.l        d0,a0
            move.l        #count,-(sp)
            move.l        #string,-(sp)
            jsr           (a0)
            add.l         #8,sp
```

```
/* from C */
int (*fastchar)() = (int (*) () )OSCONTROL(30, 0);
fastchar("string",6);
```

‘Fastchar’ returns if it encounters a control character in the string (ASCII value less than $20). It returns the number of characters actually printed in d0. Characters are drawn starting at the current cursor position, offset by the current window start. Characters are masked by the current foreground and background colours before being written into video memory. Care: blows up if asked to print beyond the last column of display.

cmd = 31   ***setbrclock***(n)
Sets a new multiplier for SCC programmable baud rate clock frequency (in Hertz). The initial value is 3,750,000. Pass a new value in n. Setting n to 0 returns the current clock frequency, without alteration. If you fiddle the clock frequency in hardware, use this to keep things functioning.

cmd = 32   ***timer1used***()
Returns true if a process is currently using the VIA Timer 1 interrupt source in the ***freetone***() system call.

cmd = 33   ***trashassign***(uid)
Removes all `assigns` made by user uid.

cmd = 34 ***trashenvstrings***(pid)
Removes all environment strings for the home shell process with PID equal to pid. If pid is -1, then every environment string is removed.

cmd = 35 ***envsub***(in, dollaronly, memmode)
Gives access to the internal environment string substitution code. It performs substitutions on the zero terminated string pointed to by in, returning a new string in d0. Space for the new string is allocated via the ***getmem***() syscall. The memory allocation mode for the new string is specified by memmode, which should be 0 or 1 (just as in ***gegmem***).

Substitutions are performed as described for `set` inbuilt command (in *1616/OS Users Reference Manual*). If dollaronly is non-zero, then only the $ type substitution is performed. Substitution is also controlled by the relevant bits in the current process's home shell's environment mode bits field.

cmd = 36 ***doassign***(argc, argv)
Low level access to the `assign` command. Argv points to a list of pointers to null-terminated strings. The syntax of these strings is the same as that required by `assign`. Argc is a count of the number of entries in argv.

7 Multitasking

Multitasking requires a range of services. These include starting asynchronous processes, scheduling new processes, providing interprocess pipes, and managing processes. Have a read through the introduction to multitasking in the *Users Tutorial Manual*.

Managing processes includes obtaining child and parent process IDs, terminating processes, causing processes to sleep or be suspended for a period, waiting for them to finish, locking processes so that they are not accidently descheduled. Sending signals, and installing signal handlers are supported. The relative time each process can obtain (***nice*** level) can be set.

Various monitor and debug routines are also available, such as providing memory usage figures for specified processes, detecting a user initiated interrupt, enabling trace mode, and determining whether a process is interactive.

A special sort of process known as a shell-type process has been introduced. The most common shell process is that which the ***iexec***() system call starts up. That is, the process which displays a prompt, reads a command from the keyboard and executes it.

Shell processes are special because they are treated as the leading process of a group of processes. An 'environment' is associated with each shell process running in the 1616. The environment consists of an inbuilt command disable vector, a group of option bits, and a collection of 'environment variables' which have various uses.

Create a communication pipe - pipe

pipe(ptr)

d0	132	
d1	ptr	Pointer to 2 longwords
Return	0 or negative error code	

This call creates a ***pipe***, typically used for communication between two processes.

Pipes are part of the file system: once created they are treated as files. One end of the pipe is written to using the ***write*** system call and the other is read from using the ***read*** system call. Other I/O system calls such as ***getchar***, ***fprintf***, etc work correctly on pipes because they all call ***read*** and ***write***.

Internally a pipe is represented as a 2 kbyte circular buffer. Writes to the pipe's input cause bytes to be added to the head of the buffer. Reads from the pipe's output cause bytes to be read from the tail of the buffer.

Since the pipe has an input and an output end, the ***pipe*** system call must return two file descriptors. The pointer ptr points to two longwords. The first receives the file descriptor for the output of the pipe: this descriptor can only be used for reads. The second longword is set to the file descriptor for the input of the pipe; it can only be written to.

A pipe is cleaned up when both its input and output ends have been closed with the ***close*** system call.

Asynchronous execa - aexeca

aexeca(argv, isasync)

d0	128	
d1	argv	Pointer to argument vector
d2	isasync	If true, run asynchronously
Return	Pid or exit code	

This system call is an upgraded version of the ***execa*** system call. argv points to a table of pointers to null-terminated strings. The table of pointers must end in a null pointer. The first pointer in the table must point to the name of an MRD, an inbuilt command or an executable disk file.

If isasync is non-zero then ***aexeca*** starts the appropriate process asynchronously and returns the process ID to the calling process. If the 0'th entry in the argv list refers to an EXEC or XREL file, it is loaded into memory, then scheduled via ***schedprep()*** before ***aexeca*** returns to the calling process.

If isasync is zero then the calling process is blocked until the desired command has exited. The exit code is returned to the calling process.

Because ***schedprep*** is used the new process inherits the current process's standard input, standard output and standard error files, as well as its current working directory and its nice value.

Programs can install new commands into ***aexeca***. These can alter the argv vector before ***aexeca*** uses it, so that aliasing and argument substitution may be performed. The installed vector can take over the command entirely, so programs can dynamically replace or act as front ends to inbuilt commands, mrds, or disk based commands.

Schedule a new process - schedprep

schedprep(addr, argv, flags, ss)

d0	131	
d1	addr	Code entry point
d2	argv	Argument vector

a0	flags	Scheduling options
a1	ss	Required stack space
Return	PID or error code	

schedprep is the lowest level of the ***exec*** system calls. It is passed the entry point of a new process and some information which is needed to prepare the process for running. The process table entries are set up and upon return the new process is ready to run.

When the system next comes to the new process's entry in the process table execution will commence at the address addr. The code at this address will find the 'nargs', 'argstr', 'argval' and 'argtype' pointers above the stack pointer in the usual manner.

On succesful completion the new process's PID is returned to the calling process.

The arguments to this system call are as follows:

addr The address at which execution is to commence.

argv The address of a list of pointers to the new process's arguments. The list is terminated by a nil pointer.

flags

Bit 0 If this bit is set the system must release the memory pointed to by addr when the new process exits. EXEC and XREL files use this. The ***aexeca*** system call loads the file into memory and calls ***schedprep*** with this bit set. When the process eventually departs the scheduler releases the memory which ***aexeca*** allocated to store the loaded program in.

Bit 1 The new process is asynchronous: if this bit is clear the current process is suspended until the new one completes. The exit code is returned. If this bit is set, ***schedprep*** returns to the parent process immediately after the new process has been prepared for execution. The new PID is returned.

Bit 2 This bit must be set if the new process is a shell type process. This flag is used by the ***proccntl*** system call in determining which process to signal when Alt Ctrl C is entered. It is also used when determining if a process is interactive or not (see ***proccntl***).

ss The amount of stack space to be reserved for the new process. 8 kbytes seems a useful figure here.

The new process inherits the current process's standard input, output and error file descriptors. Also inherited is the current directory pathname, and the parent process's nice level (see ***proccntl***).

This system call can be used to set up multi-tasking operations within a single program. For example, a modem transfer program could contain two routines; one for receiving packets, and the other for transmitting acknowledgements. Each routine is started via ***schedprep***. The transmit and receive processes communicate via common memory. They terminate by doing an ***exit*** system call, at which point the control program which started up both sub-processes can exit, allowing the system to clean up all the allocated memory.

Process management - proccntl

proccntl(mode, arg1, arg2, arg3)

d0	129	
d1	mode	0 to 41
d2	arg1	Varies, usually a PID
a0	arg2	Varies
a1	arg3	Varies

The ***proccntl*** system call provides practically all the process management functions. It has multiple quite diverse uses which are laid out in the normal system call format below. These may be considered to be many different system calls which happen to enter via the same point.

In most of the following modes `arg1` is used to represent a PID. It can do this in three ways. If `arg1` is in the range 0 to 63 it is considered to be a PID. If it is greater than 63 it is considered to be a pointer to a null-terminated string which contains either a string of decimal digits representing the desired PID number, or it contains the name of the PID, as displayed in the rightmost column of the `PS` inbuilt command.

There is an elaborate structure (defined in `process.h`) containing information about each process. Normally you would not need to work directly with his structure. This takes the form:

pid	int	process ID
name	char	pointer to its name
parent	int	parent ID
child	int	child, if waiting
idev	uint	standard in
odev	uint	standard out
edev	uint	standard error
ticks	long	how long has it run
starttime	long	when it was scheduled
whennext	uint	when to restart if sleeping
flags	short	permanent status flags
loadaddr	long	load address for file
stack	long	process stack area
stackspace	long	process stack size
stackbot	long	lowest address for stack pointer
sp	long	process current stack pointer
exitcode	int	value left from exit
cur_path	char	pointer to working directory name
timeslice	int	how many ticks it gets each time
sigvec	long	signal entry point
progsig		points to list of pending signals
padd	long	(previously sigval 12)
uid	int	process user ID inherited
umask	int	file creation mask inherited
argv	char	argument vector
homeshell	int	PID of ***iexec*** to which this belongs
pad0	char	(was user struct pointer, now unused)
alarmtime	long	when an alarm is to be sent
orig_idev	uint	stdin when process started
orig_odev	uint	stdout when process started
orig_edev	uint	stderr when process started
snoozevec	int	if non-zero, call here to see if it should run
snoozearg1	long	arguments passed to snoozevec
snoozearg2	long	
nomemerrflag	short	if set, don't signal on ***getmem*** failure
lastchildpid	int	last child we started
pg	int	process group
childtime	long	accmulated time of child
shellenv		points to shell process's environment
ipcblock		points to IPC buffers

The process flags (as reported by mode 15 ***getprocflags*** of this command) are defined as follows:

0 (1)	ps_blocked	sleeping on proc.child
1 (2)	ps_parblocked	process has blocked its parent
2 (4)	ps_exit	exit is pending
3 (8)	ps_binary	program needs text, data, bss freeing
4 ($10)	ps_killed	died via a kill
5 ($20)	ps_sigpending	signal it when it is rescheduled
6 ($40)	ps_tracetog	trace mode switch pending
7 ($80)	ps_shelproc	it is a shell process
8 ($100)	ps_nospcheck	suppress stack bound checking
9 ($200)	ps_sigblock	ignoring signals
10 ($400)	ps_bsigpending	blocked signal pending
11 ($800)	ps_stopped	stopped with SIGSTOP
12 ($1000)	ps_sigexit	signal this PID when anyone exit

The calls in detail:

Return current process ID - getpid

getpid()

d0	129
d1	0
Return	Current process ID

Returns the PID of the current process. This is constant for as long as a program runs. Lts of calls need to know h PID of the current proces. Place result of this call in a global variable early in your program, so it is easy to access (or persuade Andrew to give you a general method of having other calls return current PID when required)

Return a process's parent process ID - getppid

getppid(pid)

d0	129	
d1	1	
d2	pid	Child process ID
Return	Parent process ID	

Returns the PID of the parent process to the process whose PID is pid. Usually pid is obtained from ***getpid***.

Terminate current process - exit2

exit2(exitcode)

d0	129	
d1	2	
d2	exitcode	Process exit code

Processes which return to the system via an RTS instruction, or via system call 13 (***exit***), end up here for descheduling and cleanup. All open files associated with the current PID are closed, unless other processes are holding them open. All mode 0 memory allocated by the program is released. If necessary the memory at which the program loaded is also released.

All running processes which are children of the departing one are adopted by the startup process, '<startup>'. If the departing process was asynchronous (did not block its parent), and its parent process has installed a signal handler, then a signal is sent to the parent to indicate the death of its child process. If the departing process is synchronous, its exit code is copied into register d0 of the blocked parent process, which is restarted.

Unconditionally terminate a process - kill

kill(pid)

d0	129	
d1	3	
d2	pid	Process ID, in the format described above
Return	0 or error code	

This will terminate the designated process. If pid refers to the current process it exits. Otherwise the process is cleaned up when the scheduler revisits it. A killed process terminates with an exit code of -35.

This system call should not be confused with the KILL inbuilt command which in fact sends a signal to the process designated by pid. ***Kill*** can stop a user 0 process, even if initiated by a different user.

If pid is invalid or does not describe a currently running process, a negative exit code is returned by ***kill***. Kill now emits the 'PID: terminated' message.

Suspend processing - sleep

sleep(ticks)

d0	129	
d1	4	
d2	ticks	Number of 20 millisecond ticks
Return	0	

The current process is suspended until ticks 20 millisecond system ticks have passed.

If many processes are running the sleep may last longer than ticks.

sleep will terminate prematurely if the current process is sent a signal: programs which use ***sleep*** for timing and which can receive signals should check the current setting of the system tick count, rather than relying on sleep to delay for the correct period.

A sleeping process uses very little CPU time. 1616/OS does sleeps when waiting for character I/O, disk I/O and pipe I/O. Should make a nice alarm clock. Versions prior to 1616/OS 4.2a actually slept one tick more than desired.

Return pointer to process table entry - getproctab

getproctab(pid)

d0	129	
d1	5	
d2	pid	Process ID
Return	Process table pointer or error code	

Returns a pointer to the process table entry for the process described by `pid`. The process table entries are described in the C include file `process.h`, but will probably change with new versions of 1616/OS. Hopefully it will be possible to make the changes in a manner which simply grows the process structure, so that programs which directly read the process table entries will still work.

Change current working directory - cwd

cwd(pid, path)

d0	129	
d1	6	
d2	pid	Process ID
a0	path	Pointer to null-terminated directory path-name
Return	0 or error code	

This call was added so that the `cd` inbuilt command can change its parent process's current directory. The current directory for the process designated by `pid` is changed and 0 is returned. A negative error code is returned if `pid` is invalid. Do not allow a current process to use this syscall upon itself; use the ***chdir*** syscall instead.

Prevent/disable process descheduling - lockin

lockin(mode)

d0	129	
d1	7	
d2	mode 1	Lock
	mode 0	free
Return	None	

Calling ***lockin*** with an argument of 1 will prevent the scheduler from descheduling the calling process. Interrupts still occur, but all other processes are suspended. This can be useful in resolving certain timing races, ensuring exclusive access to I/O devices, screen/mouse drivers, etc.

Use ***lockin(0)*** to unlock the current process.

A process should be locked in for the minimum possible time. It must not do any file I/O and it must not exit, sleep or do any of the ***exec*** system calls while locked in.

Enable run statistic display - runstats

runstats(mode)

d0	129	
d1	8	
d2	mode 0	disable
	mode 1	enable
Return	None	

If ***runstats*** is enabled, the scheduler will display information about the memory usage of each process when it exits.

The stack usage is calculated by filling the process's stack area with $ffffffff longwords when it is prepared for scheduling. When it exits, the stack usage is calculated by scanning for overwriten longwords.

The allocated memory usage is calculated from memory manager tables at time of exit, and represents the amount of mode 0 memory allocated to the process when it exited. This is not necesarily the maximum amount of memory which the process used.

Displaying the stack usage figures can be used to tune the amount of memory which must be reserved for a program's stack space. See the description of the `CHMEM` program.

Wait for process to exit - wait

wait(pid)

d0	129	
d1	9	
d2	pid	Process ID
Return	0 or error code	

The current (calling) process is suspended until the process designated by pid terminates. The suspended process can still accept signals. An error code is returned by ***wait*** if pid is invalid.

Set process time slice - nice

nice(pid, ticks)

d0	129	
d1	10	
d2	pid	Process ID
a0	ticks	Time in scheduler
Return	Old nice value	

This call may be used to vary the number of system ticks for which a process runs each time it is scheduled.

The default ***nice*** setting is 1: processes are run for 20 milliseconds each. This gives good response when a number of processes are running.

When processes are scheduled they inherit their parent's nice value, so changing the nice value of the home ***iexec*** process will effectively change it for all subsequent processes. Setting a nice value of -1 for the ticks value will return the processes current timeslice, without changing it.

Send a signal - sigsend

sigsend(pid, arg1, arg2)

d0	129	
d1	11	
d2	pid	ID of process to signal
a0	arg1	Argument to process
a1	arg2	Argument to process
Return	Target process's flags	

The designated process is sent a signal. arg1 is nominally the signal type, and about 30 values are reserved: all the rest may be user defined. arg2 is also user defined. Here is the current list of signals; note the first 16 are compatible with UNIX.

sighup	1	hangup (not used by terminal driver)
sigint	2	interrupt (Ctrl C otr Break
sigquit	3	quit Ctrl \
sigill	4	illegal instruction (not reset when caught)
sigtrap	5	trace trap (not reset when caught)
sigiot	6	IOT instruction
sigemt	7	EMT instruction
sigfpe	8	floating point exception
sigkill	9	kill (cannot be caught or ignored)
sigbus	10	bus error
sigsegv	11	segmentation violation
sigsys	12	bad argument to system call
sigpipe	13	write to pipe with no-one to read
sigalrm	14	alarm clock
sigterm	15	software termination signal from kill
sigusr1	16	user defined
sigusr2	17	user defined
sigcld	18	death of a child
sigpwr	19	power fail restart (not used)
sigstop	20	suspend process
sigcont	21	restart it
sigexit	22	someone exited

Install a signal handler - sigcatch

sigcatch(vector)

d0	129	
d1	12	
d2	vector	Pointer to signal handler
Return	Previous signal vector	

This call installs a signal handler for the calling process. vector points to a routine within the calling process which is to be used to handle signals sent to this process. See the signal documentation for details. There is no signals documentation as yet.

Send ALT-^C interrupt - sendinterrupt

sendinterrupt(rootpid)

d0	129	
d1	13	
d2	rootpid	PID from which to search
Return	None	

This call is used to send an interrupt to the process which is responsible for blocking rootpid. It is used by the keyboard driver when Alt Ctrl C is pressed.

The system call descends the parent-child tree starting from the process rootpid until it finds a synchronous process upon which all the other processes are waiting. A signal (SIGINT: 2) is sent to the selected process, which should then exit.

Typically rootpid refers to the shell process ***iexec***. This call will terminate the process, returning control to the shell.

Note that only one process is signalled. If, for example, ***iexec*** has synchronously run process A, which has in turn synchronously run process B, then only process B is signalled. If process B is correctly written, or has not installed a signal handler, it will exit with exit code -35 (killed process). Process A should be written to pick this up and exit also.

Put a process into trace mode - proctrace

proctrace(rootpid)

d0	129	
d1	14	
d2	rootpid	PID to search from
Return	None	

proctrace searches down from the root process in the same manner as send interrupt above. When it finds a blocking process it puts it into 68000 trace mode: the program counter and, optionally, registers are displayed after every instruction. Calling ***proctrace*** a second time terminates tracing.

Placing a call of the type ***proctrace***(getpid()) in a program's signal handler is a good way of finding out where a program is hanging up: run the program, wait until it hangs up and then type Alt Ctrl C. It will enter trace mode and all will be revealed.

Read process flags from process table - getprocflags

getprocflags(pid)

d0	129	
d1	15	
d2	pid	Process ID
Return	Process flags or error code	

Reads the 'flags' entry from the process table entry for pid. An error code is returned if pid is invalid. See `process.h` or the start of this call description for details about the process table entries.

Determine if process is interactive - isinteractive

isinteractive(pid)

d0	129	
d1	16	
d2	pid	Process ID
Return	0, 1 or error code	

This causes a search up through the parent-child process tree until a shell-type process is found. If the process designated by pid is blocking the shell-type process then a 1 is returned by this system call. Otherwise a 0 is returned. An error code is returned if pid is invalid.

A process uses this call to determine whether or not it is running in the background: it is a background task if it or one of its ancestors was run asynchronously (with the **&** command).

A non-interactive process should produce minimal output on standard output and standard error, and should not attempt to read from the keyboard. It must run independently, not interacting with the user for directions.

Enable/disable stack checking - nospheck

nospheck(pid, mode)

d0	129	
d1	17	
d2	pid	Process ID
a0	mode 0	Supress checking
	mode 1	Re-enable checking
Return	Old value of process table flags	

Normally the scheduler checks for a process stack overrun each time a process is descheduled. Calling ***nospcheck*** with mode = 0 suppresses this checking for the process described by pid. If mode = 1, checking is reenabled.

Stack checking is important, and this system call should be used sparingly.

Install context switch vector - csvec

csvec(vec)

d0	129	
d1	18	
d2	vec	Pointer to user code
Return		Old vector setting

Causes the system to call the routine pointed to by vec every time a context switch is performed. The called code is passed the new PID at 4(sp), and a pointer to the new process's process table entry at 8(sp). Install a vector of 0 to remove.

The user code is called with the arguments as below:

4(sp)	PID being scheduled/descheduled.
8(sp)	Pointer to process table entry for process.
12(sp)	Current system tick count, from ***get_ticks***.
16(sp)	0: PID is being scheduled. 1: PID is being descheduled.

The user code may perform extended process accounting, etc. It should preserve all registers.

Pointer to current PID - getpcurpid

getpcurpid()

d0	129	
d1	19	
Return		Pointer to current PID

Returns a pointer to a long word which always represents the current process ID. This can be used for a quick ***getpid()***.

Read signal handler vector - readsigvec

readsigvec(pid)

d0	129	
d1	20	
d2	pid	
Return		Signal handler vector

Returns the address of the signal handler routine for the process described by pid. If no handler, returns zero. This call can be used to determine if a process is currently running: make pid point to a null terminated process name. If the return from this call is non-negative, the process is currently installed in the process table. Returns negative error code if PID is invalid.

File system interlock semaphore - fsbptr

fsbptr()

d0	129	
d1	21	
Return		Pointer to file system interlock semaphore.

Returns a pointer to a long word which is the file system lock-out flag. If the long word is non-zero, then a process is using the file system.

File system PID - fspptr

fspptr

d0	129	
d1	22	
Return		Pointer to file system PID.

Returns a pointer to a long word which is the PID of the process which is currently within the file system. Must be qualified by *(***fsbptr***). That is, only true while file system busy flag is also true.

Process interlock semaphore - ssptr

ssptr()

d0	129	
d1	23	
Return		Pointer to process interlock semaphore.

Returns a pointer to a long word which, if non-zero, locks the current process in. Incrementing the longword is a quick way of perfoming a ***lockin***(1). Decrementing is equivalent to a ***lockin***(0) call. Do the increment and decrement in a single instruction to avoid race conditions.

Kill user of shell - killuser

killuser(homeshell)

d0	129	
d1	24	
d2	PID of homeshell	
Return		Number of processes killed.

Unconditionally kills all processes whose home shell is homeshell, but not the actual homeshell process. A home process is one which was installed as a shell-type process in ***schedprep***, i.e., it describes an ***iexec*** type process. This system call can be used to clean up all processes started by a user, after the user logs off the machine.

Block or unblock signals - sigblock

sigblock(pid, mode)

d0	129	
d1	25	
d2	PID	Process identifier
a0	mode	1: block, 0: unblock
Return		0 or error code

Use to prevent signals being sent to the identified process.

Set an alarm - alarm

alarm(n)

d0	129	
d1	26	
d2	ticks	Ticks until alarm
Return		Time until next alarm, or error

Permits a process to request that a signal (sigalrm: 14) be sent to it after ticks 50 Hz system ticks have elapsed. If ticks is equal to -1, the number of ticks until the next alarm is returned. ***alarm(0)*** clears a pending alarm. Call returns (ticks until alarm). This value is $80000000 if there is no alarm pending.

Signal blocker - sigblocker

sigblocker(rootpid, sig, arg)

d0	129	
d1	27	
d2	rootpid	Start PID
a0	sig	Signal to send
a1	arg	Argument to pass
Return		

Sends the specified signal and argument to the process which is blocking the process identified by rootpid. The operating system descends the process table, starting from rootpid, until it locates the process which is blocking all its parents up to rootpid, and signals it. Used in the processing of Alt Ctrl C interrupt.

Sleep until true - snooze

snooze(vec, arg1, arg2)

d0	129	
d1	28	
d2	vec	Pointer to snooze function
a0	arg1	Argument passed to vec
a1	arg2	Argument passed to vec
Return		0, 1 or error

Process can sleep until a condition defined by the process becomes true. The operating system calls the user-written function pointed to by vec within the scheduling loop. When it returns non-zero, the process is rescheduled (that is, the call to ***snooze*** returns).

The arguments to ***snooze***() are passed to the user code when the system polls it.

A ***snooze***() call terminates when the vec function returns true to the scheduler, or when the process is signalled. The process's signal handler is called before the ***snooze*** call returns. ***snooze***() returns 0 if vec returns true, returns 1 if a signal is received.

User code receives arguments as below:

4(sp)	arg1
8(sp)	arg2
12(sp)	system tick count, provided by ***get_ticks***.

The user code should provide whatever tests are required, and return 0 (keep snoozing) or 1 (break the snooze). Preserve all registers, and don't try any fanch syscalls, such as file I/O, character I/O, memory manager calls, etc in your test code. PID, UID etc., are indeterminate. Example that waits for I/O port follows:

```
#include <syscalls.h>
waitforbit(ptr, mask)
unsigned char *ptr;                     /* Pointer to port */
unsigned char mask;                     /* Bit to wait for */
{
          int testfunc();
          while (SNOOZE(testfunc, ptr, mask));
                                        /* Loop until bit setting breaks it */
}

/* The scheduler calls this function */
int testfunc(ptr, mask, ticks)
unsigned char *ptr, mask;
unsigned long ticks;
{
          if (*ptr & mask)
                                        return 1;
          return 0;
}
```

Send signal to processes - siguser

siguser(pid, sig, arg)

d0	129	
d1	29	
d2	pid	PID of shell process
a0	sig	Signal to send
a1	arg	Argument to send
Return		Number of processes signalled, or error

Sends the specified signal, with specified argument, to all processes whose home shell process is identified by pid. A home shell process is not itself signalled.

Find process's home shell PID - findhomeshell

findhomeshell(pid)

d0	129	
d1	30	
d2	pid	PID whose home shell is to be found
Return		Home shell PID or error

Locates PID of the specified process's home shell process. If pid refers to a shell type process, then its home shell process PID is returned, not its own PID.

Set nice level of processes - setshnice

setshnice(pid, nice)

d0	129	
d1	31	
d2	pid	Shell type process PID
a0	nice	Time slice value
Return		0 or error code

Set time slice period (the number of ticks allocated per scheduling period) of every process whose home shell PID is that identified by pid. Pid must refer to a shell type process. The ***nice*** level of the shell type process identified by pid is not altered.

Return last child PID - lastchild

lastchild(pid)

d0	129	
d1	32	
d2	pid	Process ID
Return		Last child PID

Returns PID of the last process asynchronously started by the process identified by pid. Handy for writing reentrant code, where a single function is scheduled by the same piece of code more than one time. Returns zero if no child process is available.

Switch pending flag - swpptr

swpptr()

d0	129	
d1	33	
Return		Pointer to longword

Returns pointer to an operating system variable which, if set, indicates that the currently running process is to be descheduled as soon as it leaves a locked in state.

Kill a group of processes - killdown

killdown(startpid)

d0	129	
d1	34	
d2	startpid	PID from which to start search
Return		0 or error code

Searches down the process or child list from the specified process, unconditionally terminating all child processes. If one of these is the calling process, it is skipped.

This call is performed by the `kill -ka` inbuilt command to kill off a process, all its blocking children, and all their blocking children.

Signal a group of processes - sigdown

sigdown(startpid, val1, val2)

d0	129	
d1	35	
d2	startpid	PID from which to start search
a0	sig	Signal to send
a1	arg	Argument to pass with signal
Return		0 or error code

Like ***killdown***(), except the passed signal and argument are passed to the processes, rather than killing them.

Kill a users processes - killuid

killuid(uid)

d0	129	
d1	36	
d2	uid	User ID
Return		0 or error code

Unconditionally terminates all processes whose user ID is identified by uid. If the current (calling) process is identified in this group of processes, it is killed after all the other processes.

Signal a users processes - siguid

siguid(uid, p1, p2)

d0	129	
d1	37	
d2	uid	User ID
a0	sig	Signal to send
a1	arg	Argument to send with signal
Return		0 or error code

Semds the specified signal, with the specified argument. to all processes owned by the user identified by uid. If one of these processes is the current (calling) process, it is signalled after all the other processes in the group.

Signal process on exits - setsigexit

setsigexit(pid, mode)

d0	129	
d1	38	
d2	pid	Process to signal
a0	mode	1: enable, 0: disable
Return		0 or error code

If enabled, this call results in the identifying process being signalled whenever ANY other process exits. The identified process's signal handler is passed a sigexit(22) and the exiting process's PID. If mode is zero, signal not sent.

Set process group - setpg

setpg(pg)

d0	129	
d1	39	
d2	pg	New process group identifier
Return		New process group

A process group is a number which identifies a group of processes. It may be used to collectively identify a group of processes for the purpose of killing them or signalling them. Normally all processes run with a process group ID of 2. This call allows a process to modify its process group number. All processes inherit their parent's process group number when they are created.

If pg is 1, the current process's group is set to a new value, which is initially 3, then 4 with the next call to ***setpg***(), etc.

If pg is zero, then the current process's process group number is unaltered. A process may use this to find its current process group number.

For all other values of pg, the current process's process group number becomes pg.

The new process group number for the current process is always returned.

Signal processes in a process group - sigpg

sigpg(pg, sig, arg)

d0	129	
d1	40	
d2	pg	Process group number
a0	sig	Signal to send
a1	arg	Argument to send with signal
Return		0 or error code

All processes with the specified process group number are signalled with the passed signal, and signal argument. If the calling process belongs to that group, it is signalled last.

Kill processes in a group - killpg

killpg(pg)

d0	129	
d1	41	
d2	pg	Procss group number
Return		0 or eror code

All processes with the specified process group number are unconditionally erminated. If the calling process belongs to that group, it is killed last.

Set group file creation mask - setprocumask

setprocumask(pid, umask)

d0	129	
d1	42	
d2	pid	ID of process to alter
a0	umask	New umask
Return		New umask

Sets the specified process's file creation mask to umask. Returns new umask. If umask is -1, the current file creation mask is returned unchanged.

Manipulate environment bits - setenvbits

setenvbits(pid, mask, mode)

d0	129	
d1	43	
d2	pid	Process ID

a0	mask	Bit mask for environment bits
a1	set	0: clear, 1: set, 2: ea
Return		New environment bits

Manipulates environment bit field attached to a shell type process. See the shell process documentation.

Locates the home shell process of the PID identified by `pid`, and modifies its environment bits. This changes the environment of that shell process, all processes which have it as their homwe shell, and all child processes of it.

If `set` is zero, set bits in `mask` are used to clear bits in the environment (that is, `mask` is inverted and ANDed into the environment bits).

If `set` is one, `mask` is ORed into the environment bits.

If `set` is two, the environment bits are read. In all modes, the environment bits are returned. The new program `setenv` is provided for manipulation of these bits.

Every time this call is performed, the system records the resulting environment bit settings, and uses it to start off the root shell process when the system is next reset. Thus you don't lose your environment upon reset.

The environment bits are described in `envbits.h`, and are as follows:

0 (1)	envb_prompten	dir name in prompt
1 (2)	envb_verbose	run in verbose mod
2 (4)	envb_dirmode0	no directory sorting
3 (8)	envb_dirmode1	0 sort dir by date and time 1 sort dir alphabetically
4 ($10)	envb_ibbeep	Beep on inbuilt command error
5 ($20)	envb_nobak	Editors don't generate `bak` file
6 ($40)	envb_errbeep	Beep when programs exit with error
7 ($80)	envb_hidefiles	Hides files with hidden bit set
8 ($100)	envb_assignprompt	Truncate prompt with assignments
9 ($200)	envb_lowernames	Enable lower case filenames
10 ($400)	envb_promptgt	Enable '>' in prompts
11 ($800)	envb_dodsign	Do '$' exec environ substitutions
12 ($1000)	envb_doarg	Do argument environ substitutions
13 ($2000)	envb_doarg0	Do argv[0] environ substitutions
14 ($4000)	envb_noshellout	This user cannot shell out

The default evironment settings are verbose mode commands, directory and > in prompt, alphabetically sorted directories, beep on inbuilt command errors, lower case filenames, $ and argv[0] environment substitutions.

Read environment bits - getenvbits

getenvbits(pid)

d0	129	
d1	44	
d2	pid	Process identifier
Return		Environment bits or error

Returns the environment bits of the home shell process of the process identified by pid.

Convert string to PID - nametopid

nametopid(pid)

d0	129	
d1	45	
d2	pid	PID identifier
Return		PID number or error

Converts its argument into a process ID. If pid is a number in the range 0 to maxpids (currently 64), it is simply returned. If pid points to a null terminated string of digits, the string is evaluated as a decimal number and returned. If pid points to a null terminated name of a process, the process table is searched for a process with a matching name. The PID of the first one encountered is returned.

Send an IPC block - blocktx

blocktx(destpid, addr, length, sig)

d0	129	
d1	46	
d2	destpid	Pid to send block to
a0	addr	Address to send block to
a1	length	Length of block to send in bytes
a2	sig	0: don’t signal, 1: signal destpid
Return		0 or error code

Sends a block of data to the identified PID. See ***blockrx*** for more details of use.

Receive an interprocess block - blockrx

blockrx(mode)

d0	129	
d1	47	
d2	mode	Mode of operation
Return		Varies

Processes use this call to manipulate their interprocess block communication input queue. There are five modes of operation:

ipcb_gethead (0)	Return pointer to head of list
ipcb_blockcount (1)	How many blocks are queued?
ipcb_bytecount (2)	How many bytes are queued?
ipcb_ftchnext (3)	Return next block, 0 if none.
ipcb_fetchwait (4)	Get next block, sleep until available.

The input queue hangs off the receiving process's process table entry, and consists of a linked list, in the order of receipt. The structure of the linked list is:
Pointer to next structure
whofrom, the sender PID (long)
blocklength, the size of the data block (long)
padd, six longs for future expansion
data, variable size, as unsigned chars

Manipulate environment strings - setenv

setenv(pid, name, setting, mode)

d0	129	
d1	48	
d2	pid	Process ID
a0	name	Name part of env variabl
a1	setting	Setting part of env variabl
a2	mode	Mode of env string substitution
Return		0, 1 or error

Permits alteration of the environment variable strings which are attached to the home shell process of the process identified by pid. These strings are shared by the home shell and by all processes which share that home shell.

Name points to a null-teminated string of any length which identifies the environment string. Setting is the new string which is associated with the name string. If setting is zero (a nil poiner), then the environment string identified by name is remved.

Mode determines how the new name and setting are to be handled by the ***envsub***() system call, which is usedin command line substitution. It is a combination of the following bits, defined in `envbits.h`.

envs_dsign	1	Enable '$' type usage
envs_arg	2	Enable substitution in all arguments
envs_arg0	4	Enable substitution at start of command

Note that enabling of substitution is also controlled by bits in the home shell's environment bit.

This call returns 0 if all worked, and `name` did not previously exist in the environment. It returns 1 if all went well, and `name` was previously defined in the environment. A ngative eror code is returned on memory allocation failure, bad PID, etc.

See the documentation on shell type processes, and the `set` inbuiltcommand for more details.

Environment string variable - getenv

getenv(pid, name, mode)

d0	129	
d1	49	
d2	pid	Process identifier
a0	name	Pointer to name string
a1	mode	Search mode
Return		0, error code, pointer to string

Provides access to the environment string variables attached to the home shell process of the process identified by `pid`.

`Name` points to a null-terminated string which identifies an environment variable. The string comparison is case sensitive, so be careful.

`Mode` is a mask, consisting of a combination of ENVS_DSIGN, ENVS_ARG and ENVS_ARG0. If the result of ANDing `mode` with the environment variable's mode (as passed when it was installed) is non-zero, then it is valid.

If no match is found, a value of zero is returned. If a matching string is found, and the ANDing of `mode` and the environment variable's mode succeeds, this call returns a pointer to the variable's 'setting' string.

If `name` is less than $4000, then it is assumed to be a numeric index into the environment variables. If `name` does not exceed the number of curently installed environment strings, then a pointer to the corresponding ENVSTRING structure (defined in `process.h`) is returned. The `mode` field is ignored in this mode. If `name` exceeds the number of installed environment strings, then a nil pointer is returned.

Index

Table of Contents