1616: Programmers Manual

Version 4.089

August 1993

Applix 1616 microcomputer project
Applix pty limited

1616 Programmers Manual

Even though Applix has tested the software and reviewed the documentation, Applix makes no warranty or representation, either express or implied, with respect to software, its quality, performance, merchantability, or fitness for a particular purpose. As a result this software is sold "as is," and you the purchaser are assuming the entire risk as to its quality and performance.

In no event will Applix be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or its documentation.

The laws in some countries or states may modify the effects of the disclaimer above.

The original version of this manual was written by Andrew Morton
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 Applix BBS (ringback) on (02) 554 3114 and (02) 540 3595

ISBN 0 947341 01 3

MC68000®™ is a trademark of Motorola Inc.

1
Programming under 1616/OS v4

Here we describe the tools available to programmers for developing stand alone application programs to run on the 1616 under 1616/OS. In particular, the 1616/OS EPROM code entry points, the ‘system calls’, or ***syscalls*** are individually documented, and the standards and conventions which **should** be obeyed by a 1616 program are described.

The descriptions of the system calls are written from an assembly language viewpoint, with a C programming language flavour. This documentation is applicable to assembly language programs, and to any high-level language for which an assembly language interface is available. This does not imply that assembly language is essential; most Applix programs, including the operating system itself, are written in C.

For best results, the reader should be familiar with either 68000 assembler language, or C programs (preferably as used in Unix) or at least should read an introduction to 68000 assembler, prior to attempting to make extensive use of the material in this manual. The material presented herein does assume some background knowledge, and the beginner must be prepared to do some study prior to making extensive use of the system calls. By implication, the reader should already have some experience in the use of a programming language. A complete C tutorial, including manual, source code, and code examples on disk, is available from Applix.

There are some demonstration assembly language programs on the cassette tape or floppy disk you received with your 1616. These contain examples of the use of system calls, together with trap macro files.

In addition, the Users Disk for each version of the operating system contains additional documentation on disk, describing many of the updates. In particular, you will generally find an extensive range of C program header files. These header files should be used when accessing the system calls, as they are consistent with the documentation provided (this manual, for example, is heavily based upon the files provided on disk). In general, the header files in the `include` directory can be considered the most up to date authority on how to use the system calls. Files provided for programmers include:

blkerrcodes.h	the error codes
blkio.h	block and file input and output, multiblock
cassette.h	cassette leader and lock information
chario.h	Character file descriptors, chardriver structure
chdev.h	`chdev` program and ***cdmisc*** syscall header
datetime.h	structure and storage of date and time
envstring.h	user environment structure and control bits
files.h	file manipulation structures, FCB, status bits
hwdefs.h	hardware locations, interrupt definitions
mondefs.h	monitor front end and interpreter
options.h	defs for the ***option*** syscall
process.h	group process control structure, bits in proc flag
reloc.h	relocatable file format and memory resident driver
scc.h	constants and structures for serial I/O
signal.h	signal definitions, a la Unix
spspace.h	default stack spaces
ssdd.h	disk controller locations, commands, etc
storedef.h	needed by `scc.h`
syscalls.h	all the system calls, by number (use this!)
types.h	needed by `sc.h`
via.h	6522 VIA simulated interrupt vectors and locations
windows.h	video window structure, plot modes, mouse

Additional information on low level system calls, memory resident drivers, process control and multi tasking is available in the *Technical Reference Manual*, which forms the logical extension of this manual.

Transient programs

Transient programs load from disk. In 1616/OS, unlike Unix, the MC68000 is always in ‘supervisor’ mode, unlike UNIX. There is little point in using ‘user’ mode in a system of this kind.

Transient program conventions

It is desirable that general transient programs blend into the 1616/OS environment by working in a standard, predictable manner. Conventions which should be observed are:

- Programs should, where possible, obtain their user-provided information from the command line arguments, using the argc and argv constructs available in C, as with the 1616/OS inbuilt commands. Programs are **tools**, and are used in combination with other programs to build further tools. In particular, use the argv and argc constructions available under C.

  Programs which interactively prompt the user to enter filenames, answer questions and the like are a nuisance to drive from within shell files, particularly where argument substitution is used, and they cannot be successfully chained to from other transient programs.

The operating system is multitasking; where possible ensure programs can be run in background mode, without interaction with the user. Allow input redirection by using standard input, standard output and standard error. Avoid direct access to the hardware. Specific details on multitasking and multi user programming appears in the *Technical Reference Manual*.

Write your command argument processing so that the program is insensitive to the order in which arguments appear on the command line.

- If a program detects an error on the command line it should print out a usage message to standard error, of the form (for example):

  ```
  File copying program Version 2.1, XYZ Pty.Ltd.
  Usage: tpfilename infilename outfilename [-n]
  ```

  Note that the function of the program, its version number and a usage format message which describes how to use the program are all printed out. All optional arguments in the usage message are put in square brackets. When printing out a usage message use the first entry in the 'argstr' array (described later) to represent the name of the program. This way you know what the user of your program is calling it, rather than assuming that he uses the same name for it as you did when you wrote the code.

- Do not produce unnecessary output! Programs which display copyright messages, programmer's names, version numbers, etc. are often awkward to use in an operating system with the I/O redirection features of 1616/OS. Often all you wish from a program is its output, which can be redirected onto a temporary file and then redirected onto the input of another program. Undesired output can get in the way of all this. Version numbers, etc. are useful; put a '-v' flag into your program for verbose mode, as with the 1616 assembler, SSASM.

- Direct error messages to the standard error stream (handle $100). This provides consistency and permits the user to direct error messages to a printer, file, etc. without them getting mixed with normal output.

Transient program command line arguments

There are two types of executable program files supported under 1616/OS. The original type are `exec` files, indicated by a `.exec` at the end of their filename. These are absolutely located programs. Their load address (and entry point) appear under 'LOAD' in the directory listing. This style of program will probably not be supported in future operating system releases, so all programs should be converted to `.xrel` form prior to release.

The standard start address for `.exec` files is $4000, however programs may be loaded in at higher addresses. When a transient program is invoked from the command line the following steps occur:

- If it is an `.exec` file, the system loads the program into memory at the address indicated by the file's 'load address' field. If it is an `.xrel` file, the system allocates memory for the file, loads and then relocates it.

- Four longwords which contain information about the command line arguments are pushed onto the MC68000's stack. See below for details
- 1616/OS transfers control to the loaded program by performing an MC68000 'JSR' to its load address, hence the start of the file must be the entry point.
- The transient program executes. When it has finished it returns control to 1616/OS via the MC68000 'RTS' instruction, so the transient program must maintain stack discipline. The value which the application program returns in data register zero (d0) is the return value to 1616/OS and to any program which has run this one. A non-negative return value indicates no error. A negative return (bit 31 set) indicates an error.

The four long words which are present on the stack, upon entry to transient programs, permit convenient access to the command line arguments which were entered when the program was invoked. They are compatible with the 'argc/argv' construct which is available to C programs under UNIX, CP/M, MSDOS, etc.

The long words on the stack are:

Name Address Use

nargs 4(a7) This is the number of command line arguments, including the name of the transient program file. nargs will always be at least 1.

argstr 8(a7) argstr is a pointer to the first entry in an array which contains nargs pointers, each of which points to null-terminated strings. The first pointer in the array points to the first command line argument, etc.

argtype 12(a7) argtype is a pointer to an array of nargs bytes, each of which specifies the type of the corresponding command line argument. The bytes can have one of two values:

$02: A numeric argument
$04: A non-numeric argument

This array may be used in conjunction with nargs and the argval array for accessing numeric data on the command line.

argval 16(a7) This is a pointer to the first entry in an array of long words, each entry of which is a binary representation of the corresponding number on the command line. This array must be qualified by the argtype array: if the n'th entry in the argtype array is not $02 (numeric argument) then the n'th entry in the argval array is undefined. Use the argval array wherever possible: it converts hexadecimal, binary and decimal numbers and promotes a standard transient program argument format.

The second type of program file is the `xrel` file; these are relocatable programs which may be loaded and run at any address in memory. `.xrel` files are preferable to `.exec` files. See the documentation on relocatable files for more details.

A relocatable file consists of a header, the code and data (executable at zero), and the relocation information. The header has the following structure.

magic1	ushort	magic number $601a (actually a BRA)
textlen	int	text length
datalen	int	data length
bsslen	int	BSS length
symtablen	int	symbol table length
stackspace	int	unused
text_begin	int	start address
reloc_flag	ushort	relocation bits follow

If the relocation flag is non-zero, then relocation information follows. This consists of a longword, which is the offset into the loaded code of the first relocatable longword. If zero, then 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. All byte offsets are even numbers. The byte $01 means ‘add decimal 254 to the location pointer, without performing a relocation’. This covers cases where two neighbouring relocatable longwords are more than 254 bytes apart. The entire sequence is terminated with a byte of $00.

The `relcc` C preprocessor produces relocatable programs from C. From assembler, there is a shell file on the *Users Disk* that will produce a relocatable program by performing two passes of the assembler.

Transient program memory model

A transient program cannot use any memory without requesting it from the system memory allocator. See the memory manager section of the *Technical Reference Manual* for details.

The memory range $0 to $3FF is reserved for 68000 vectors and some system use (described later).

The memory range $400 to $3BFF is reserved for 1616/OS usage.

The range $3C00 to $3FFF is used for copying in the boot block from a disk device whenever the 1616 is reset.

I/O addresses

The 1616's I/O devices are memory mapped. Their addresses are given here mainly for reference purposes; if possible you should use the available system calls for I/O. All devices are one byte wide and are addressed as follows:

Address	Mnemonic	Usage
$600001	centlatch	Centronics (parallel printer) latch
$600081	daclatch	D/A converter latch
$600101	vidlatch	Video latch
$600181	amuxlatch	Analogue multiplexor latch
$600000	pal0	Video palette entry 0
$600020	pal1	Video palette entry 1
$600040	pal2	Video palette entry 2
$600060	pal3	Video palette entry 3
$700000	sccbcont	SCC channel B control register
$700002	sccbdata	SCC channel B data register
$700004	sccacont	SCC channel A control register
$700006	sccadata	SCC channel A data register
$700081	iport	The input port
$700100	viabase	VIA base address. VIA registers start at this address and appear at every second byte address.
$700180	crtcaddr	MC6845 CRTC address register
$700182	crtcdata	MC6845 CRTC data register

Shadow registers

The four latches and the video pallette are write-only. We need to know their current contents if we are to alter only some of their bits. There are also other hardware setting that are more convenient if recorded in a fixed memory location. For this reason there are a number of bytes called shadow registers which contain the current contents of the latches and pallette. The shadow registers should be updated when the latches and pallettes are changed by direct access to the hardware.

The shadow registers are:

Address	Mnemonic	Usage
$300	vlval	image of video latch
$302	clval	image of centronics latch
$304	dlval	image of D/A converter latch
$306	alval	image of analogue multiplexor latch
$308	palval0	image of pallette entry 0
$30A	palval1	image of pallette entry 1
$30C	palval2	image of pallette entry 2
$30E	palval3	image of pallette entry 3
$310	ak_ctrl	control key depressed if non-zero
$312	ak_shift	Bit 0 left shift down Bit 1 right shift down
$314	ak_alt	Alt key down
$316	ak_capslock	Capslock is active
$318	ak_numlock	Keypad in numeric mode

There are some timing considerations which must be observed if we are to avoid writing one of those programs which rarely but regularly fails. If you must bypass the system calls and directly write to a latch or the palette, write to the shadow register first. This means that if a higher priority interrupt routine catches your code between the two writes, the desired byte will still reach the latch. One general cure to this timing problem is to temporarily raise the processor interrupt priority during the alterations, to prevent any other code from interfering.

Simulated interrupt vectors

The interrupts are autovectored on the 1616 and the SCC and the VIA do not support multiple interrupt vectors, so 1616/OS simulates multiple interrupt vectors for these devices. When the VIA or the SCC interrupts for any reason, 1616/OS ascertains from the device the reason(s) for the interrupt, and vectors through one or more of the following addresses:

(ISR stands for ‘interrupt service routine’)

$100	Pointer to VIA timer 1 timeout ISR
$104	Pointer to VIA timer 2 timeout ISR
$108	Pointer to VIA CB1 ISR
$10C	Pointer to VIA CB2 ISR
$110	Pointer to VIA shift register ISR
$114	Pointer to VIA CA1 ISR
$118	Pointer to VIA CA2 ISR
$140	Pointer to SCC channel A character receive ISR
$144	Pointer to SCC channel A character transmit ISR
$148	Pointer to SCC channel A external/status ISR
$14C	Pointer to SCC channel B character receive ISR
$150	Pointer to SCC channel B character transmit ISR
$154	Pointer to SCC channel B external/status ISR

It is the responsibility of each called interrupt service routine to clear the source of its interrupt (and no others) from the interrupting device.

Interrupt priorities

1616/OS requires that the 1616’s interrupting devices be set at the following priorities:

Cassette IRQ	Level 4	$70
SCC IRQ	Level 3	$6C
VIA IRQ	Level 2	$68

If you mask off some or all of these interrupts by raising the processor priority, do it for as short a time as possible because the keyboard, vertical sync interrupts, date/time drivers, cassette, sound and serial communications are all interrupt driven.

The system never puts the interrupt priority over 7. Note that *Minix* uses interrupt 5 ($74) for disk interrupts.

All interrupts on the 1616 are autovectored, so the interrupt vectors are in the range $64 (level 1) to $7c (level 7)

Video colours

The 16 video colors are selected by writing 4-bit nibbles to either the video RAM (in 320 column mode) or to the palette (640 column mode) or to the video latch for the borders (both modes).

On a monochrome monitor the brightness should increase with increasing nibble value, with a value of 0000 corresponding to black.

The colours map as follows:

Number	Colour	Mnemonic
0	Black	PC_BLACK
1	Dark grey	PC_DGREY
2	Dark blue	PC_DBLUE
3	Mid blue	PC_MBLUE
4	Dark green	PC_DGREEN
5	Green	PC_GREEN
6	Blue grey	PC_BGREY
7	Light blue	PC_LBLUE
8	Dark red	PC_DRED
9	Red	PC_RED
10	Dark violet	PC_DVIOLET
11	Violet	PC_VIOLET
12	Brown	PC_BROWN
13	Yellow	PC_YELLOW
14	Light grey	PC_LGREY
15	White	PC_WHITE

Memory Layout

The organisation of memory is determined to some extent by the contents of the MRDRIVERS file, or by the setting of the DIP switches on the 1616 motherboard, so precise addresses cannot be given here. If you have an additional memory card, all the addresses above the stack space will be increased by at least one megabyte.

$080000 512k	Video Page	Top of memory
$078000 480k	11k approx bitmap buffers	Top of bitmaps
$075400 470k	RAM disk	End of RAM disk
	Memory resident drivers	End of MRDs
	Stack builds down from here Stack space	Top of stack
	Memory allocator builds down from here Free Memory `.exec` programs build up from $004000	Top of allocated memory
$004000 16k	1k area for boot block	`.exec` file start
$003C00 15k	1616/OS BSS segment	
$000400 1k $000000	System vector table	Start of memory

The size of the video page, RAM disk, MR drivers, default pallette colours, background colour, and the stack space are determined by the contents of the MRDRIVERS file which is loaded at boot time. If you have a memory expansion card, the video is at least one megabyte higher than shown. Video is at the very top of memory, and can be any multiple of 32k, up to a half megabyte. You can have multiple video screens (there is a syscall ***set_vdp*** for swapping from one display screen to another, and another called ***set_vap*** for changing which screen will be used for updating the video - thus you can update a screen independent of the display of another).

The bitmap buffers are reserved for buffering the bitmap blocks of /RD, /F0, /F1, /H0 and /H1. The 1 kbytes typically used for each disk drive bitmap is sufficient for an 8 megabyte volume, however /H0 typically allows for 40 megabyte.

EXEC file problems

There is one huge problem with this memory layout. The stack space must be fixed and relatively small. The whole philosophy of the operating system emphasises recursion of ***exec***s - the system itself performs ***exec***s in a number of places. The spawning of programs from within other programs tends to use up stack space quickly, and there is no effective way of preventing stack overruns.

The obvious solution to this problem is to build allocated memory **up** from $4000, rather than down from from some point some arbitrary distance beneath the stack. This way the heap and the stack move towards each other, and chaos comes only when all the system's memory is used. The only reason this has not been done is the need to support `.exec` files. These load at $4000, and can be of any length.

Future releases of 1616/OS will probably not support `.exec` files! The allocator will build the heap upwards towards the machine stack, and the only executable binary file format supported by 1616/OS will be `.xrel` files.

Boot Sequence

The following things happen when the system is reset. Those steps marked † only occur at level 0 resets.

- The stack pointer is initialised to $10000
- The RAM system call vector table is initialised.
- All I/O devices and major internal program modules are initialised. The 6545 CRT controller is initialised prior to using any code that requires RAM in operation.
- A small model memory manager is installed, using the $4000 - $10000 area as free memory.
- Default values for the RAM disk size, stack space and video RAM space are installed.†
- A search is made on the /F0, /F1, /H0 and /H1 drives (if present) for the MRDRIVERS file. If found, it is loaded in and new values for the RAM disk size, stack space and video RAM space are installed. The memory resident driver code is loaded in and relocated.†
- The stack pointer is moved to point to the area just below the MR drivers.
- Interrupts are enabled.
- A search is made from address $800000 through to $FFC000 in $4000 byte increments for a ROM with the values $12, $B5, $06, $A7 at the start. For each external ROM with this pattern at the start, the OS performs a JSR to the start of the ROM + 4. A zero is passed at 4(sp) and the reset level at 8(sp). At this point external ROMS can perform whatever initialisation is necessary for their purposes.

- The previous step is repeated, except a value of 1 is passed at 4(sp) to any called ROM code. It is at this point that an external ROM can take control of the system, with all of the normal system resources available.
- Every memory resident driver is called with command number 0, 1 or 2, depending upon the reset level.
- A search is made of the /RD, /F0, /F1, /H0 and /H1 devices for a bootable device. If one is found (BOOTBLOCK field of the root block non-zero), the boot block is read from the device into memory at $3C00. The system then performs a JSR to address $3C00, passing the reset level at 4(sp), and the number of the boot device driver at 8(sp). The /RD driver is device 0, /F0 is device 1, etc.
- The system drops into an infinite loop, performing ***iexec*(1)** system calls.

2
The System Calls

Broad categories

For the purposes of description the system calls are divided into the following broad categories:

System control

These functions alter the configration of the 1616 or call general purpose internal 1616/OS routines.

File and block I/O

These functions include all the control routines as well as block device driver calls for disks.

Character I/O

The character I/O functions perform input, output and status calls on character device drivers.

Video output

The video control functions include the simple calls to change colours, etc as well as the text/graphics window control functions.

Graphics calls

The graphics functions are related to the video functions, particularly where windows control functions.

Multitasking

Support for multitasking, pipes, signals, asynchronous, and multiple user processes, with a generally UNIX like flavour. These are described in the *Technical Reference Manual*.

Hardware control

There are a number of system calls which manipulate I/O devices, include analog and digital converters, etc. Use these wherever possible, rather than directly manipulating I/O devices.

The system call mechanism

A system call is performed by putting the call number into d0 (data register zero). Any required arguments go into d1, d2, a0, a1 and a2. Then execute the 68000 ‘TRAP #7’ instruction. If the system call does not require five arguments, then not all of these registers need be initialised. Appropriate header files are provided for both assembler and C programmers, with an emphasis on C.

Any return value from the system call will be in d0. If the system call does not return a value, then the contents of d0 are undefined.

All system calls preserve all registers except d0, however it is poor programming practice to rely upon this.

All parameters passed to system calls are considered to be long integers (32 bit quantities). The returned value is also a long. MS-DOS programmers, and those converting programs, should take care not to be caught by the difference in ints (integer numbers are 32 bits in 68000, 16 bits in 8086).

As an example of a system call let us consider a subroutine to print out a byte in binary, decimal, hexadecimal and ASCII format. In this example we use the ***printf*** system call to do the printing.

```
*
* Example subroutine to print out a byte
* (at ‘num’) in 4 ways.
* printf control string
*
print4    move.l      #control,d1
          clr.l       d2
          move.b      num,d2      * number to print
          move.l      d2,a0
          move.l      d2,a1
          move.l      d2,a2       * 4 copies
          move.l      #48,d0      * printf syscall no.
          trap        #7          * do the call
          rts                     * no meaningful return

control   dc.b        "bin: %b, dec: %d, hex: %x, ascii:
%c",13,10,0
```

Here the parameters required by the system call are loaded into the appropriate registers, the call number is put in d0 and the trap is performed.

A ‘pointer’ to a data structure is a 32 bit number which is equal to the address of the first element in that structure.

Since 1616/OS is written in the C programming language, strings are invariably null-terminated. This means that the end of a character string is denoted by a zero ($00) byte at the next address beyond the last character of that string.

The Line A Trap

1616/OS versions 3.0 and later support a second mechanism for performing a system call. Although it is provided, this method should not normally be used by 1616 programs. The normal system calls should be used wherever possible.

The Line A trap system call mechanism involves pushing all the arguments to the system call onto the stack, as longwords, in reverse order. Then execute an opcode computed from ($A000 + system call number). It is the calling code’s responsibility to adjust the stack pointer for the pushed arguments.

The Line A mechanism is mainly for internal use by 1616/OS, but may be used by other programs. Registers d0, d1, a0 and a1 are trashed. All others are preserved. It does have the disadvantage that programs which use it will not run under earlier versions of the OS.

The system returns from a Line A trap in supervisor mode. The normal ‘trap #7’ preserves the User/Supervisor state.

The system call example presented above can be reworked to use the Line A trap as presented below.

```
*
* Example subroutine to print out a byte
* (at ‘num’) in 4 ways
* using the Line A system trap
*
* printf system call number
r_printf  equ         48
print4    clr.l       d2
          move.b      num,d2      * number to print
          move.l      d2,-(sp)
          move.l      d2,-(sp)
          move.l      d2,-(sp)
          move.l      d2,-(sp)    * pass it 4 times
          move.l      #control,d0 * printf control string
          move.l      d0,-(sp)
          dc.w        $A000+r_printf  * off to the OS
          add.w       #20,a7      * adjust 5 longs
          rts                     * no meaningful return
```

Format of the system call documentation

Due to the number of system calls and the cost of paper it is necessary to define a brief format for describing the system calls. The general format is:

callname(arg1, arg2)		Brief description
d0	call number	
d1	arg1	usage of d1
d2	arg2	usage of d2
a0	arg3	usage of a0
a1	arg4	usage of a1
a2	arg5	usage of a2
Return		description of returned value.

Where the ***callname*** is the standard name of the system call and arg1, arg2, etc., are the names of the arguments which the call requires. As the arguments are passed in registers, their use is described in the lines after the call definition, along with which register they must go in.

To make the manual clearer, syscalls are shown in ***bold italic***, while arguments are shown in helvetica. Example code, and 1616/OS commands, are shown in `courier`.

The ‘Return’ value is a description of what (if anything) the system call returns in d0. Many system calls return a negative number (bit 31 of d0 set) if an error of some nature is detected. The meanings of all known error messages are listed in Appendix B, together with their decimal and hexadecimal value. Remember there are two syscalls (114 and 122) to interpret error messages, and the ***printf*** syscall will also print English versions from the error codes by using the %e option. Do not rely upon simply testing for -1 as an error, as numerous other codes are available. Test for a negative number.

3
System Control Calls

Introduction

These are general purpose utilities, and system control calls. They includes several levels of reset, exiting a program, interrupt and VIA routines, raw cassette routines, time and date facilities, calculating sines, allocating and releasing memory, various line editor entry points, altering the keyboard scan, a quicksort, and various ***exec*** routines.

Reinitialise 1616/OS - coldboot

coldboot( )

d0	101

Restarts 1616/OS as if the system had just been turned on at the power switch (some early versions used 0 for this syscall, but 0 is now a ***warmboot***).

Reinitialise 1616/OS - warmboot

warmboot( )

d0	1

Restarts 1616/OS as if the reset button or ALT-control-R had been pressed.

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. See *Technical Reference Manual* for details.

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.	

This system call loads a program (`.exec` or `.xrel`) into memory. See *Technical Reference Manual* for details.

Terminate a transient program - exit

exit(retval)

d0	13	
d1	retval	Program return value
Return	??	

When the system has loaded a program from disk and is about to execute it, all of the 68000 registers except D0 are saved. This includes the stack pointer. If the loaded program performs the ***exit*** system call, the stack and other registers are restored and the program exits. The value in D1 at the time of the ***exit*** becomes the program's return value, much the same as the value in D0 when a program ends in the normal way with an RTS instruction.

This call is provided for an emergency way out of a program when an irretreivable error is detected when the program is several layers deep in subroutines. Be warned that it will lock the system if there is no transient program from which to exit.

The ***exit*** only applies to the currently running program. If program A ***exec***'s program B and program B ***exit***s then control is returned to program A, with retval in D0 (the normal place for a return value from a system call).

Install a vertical sync interrupt routine - set_vsvec

set_vsvec(vec, rate, callval)

d0	16	
d1	vec	Pointer to subroutine
d2	rate	Call period (50ths of a second)
a0	callval	Value passed with call
Return	vector number (-1 on error)	

This call installs in a table a pointer to an interrupt subroutine (ISR) which you have written. Your routine is called at a frequency of 50/rate Hertz. The callval parameter is one which you specify when installing the vector; it is passed to the ISR each time the ISR is called.

Before your ISR is called the following long words are pushed onto the system stack:

4(a7)	Value returned in d0 from previous call to your code
8(a7)	callval, specified when vector was installed

Up to 32 vertical sync interrupt routines may be installed. The system uses 6 of these. If there are none free in the table a value of -1 is returned; otherwise the index into the vector table is returned. This must be saved for deleting your vector at a later stage.

Your interrupt subroutine must preserve all registers (except d0) and end with an `rts`.

If this call is used by a transient program the vector must be removed before the program returns to 1616/OS. Failure to do this will result in the next transient program overwriting still active interrupt code. Thus will probably crash the system.

All entries in the vector table are cleared at any level of reset.

`set_vsvec(1, nn, 1)` (new in V4.2a) returns a pointer to a data structure that represents the state of vertical sync vector number *nn*.

Remove a vertical sync interrupt routine - clr_vsvec

clr_vsvec(vnum)

d0	17	
d1	vnum	Vertical sync vector table number
Return	0 or -1	

Removes a vertical sync ISR vector table entry. vnum is the index returned by ***set_vsec*** (above). Returns -1 if vnum is bad. May be called from within a vertical sync ISR (to remove your ISR, for example).

Get number of ticks since system startup - get_ticks

get_ticks( )

d0	18
Return	count

Returns the number of 50 Hertz ticks since the last level 0 reset - probably when the 1616 was turned on.

Determine the current CPU type - get_cpu

get_cpu( )

d0	19
Return	0 for 68000 1 for 68010

1616/OS supports both the MC68000 and the MC68010 processors. This system call is provided for programs to determine the type of CPU the system has. It returns 0 for a plain old 68000, 1 for a 68010, and won’t return 2 for a while yet.

Unfortunately, since Andrew doesn’t normally run a 68010 (they are expensive, and don’t run at 15 MHz), the 68010 support sometimes breaks on new releases of 1616/OS. We recommend sticking with the 68000 at the moment.

Raw cassette block write - caswraw

caswraw(start, length, leader)

d0	21	
d1	start	Block start address
d2	length	Block length (in bytes)
a0	leader	Leader length in bytes
Return	nil	

Writes the specified block of memory out to tape in a single stream of data followed by a checksum. The leader argument is the number of $ff bytes to use for a leader; use 800 decimal here.

The resulting output is not compatible with 1616/OS cassette files, however it may be read in using the ***casrraw*** system call. This can be used to test the cassette interface.

Raw cassette block read - casrraw

casrraw(buf, leader, maxhunk)

d0	22	
d1	buf	Block start address
d2	leader	Lock-in length
a0	maxhunk	Maximum block size
Return	block length (-1 on error)	

Reads a raw block from the tape into memory. The lock-in length is the minimum number of $ff bytes required for a leader lock; use 250 decimal here. Since the size of the tape block is not known prior to reading you must specify the largest size acceptable, maxhunk.

Returns the length of the block if sucessful, or -1 if checksum error or blocksize > maxhunk.

Get system time date - getdate

getdate(buffer)

d0	23	
d1	buffer	date/time buffer
Return	buffer address	

Moves the current date/time to the seven byte area pointed to by buffer. The data at (buffer) is: year, month, date, hour, minute, second, tenths of seconds. Since we use a single byte for the year, the preceeding 19 is assumed in the year. This will probably have to be fixed in 1999.

Set system time date - setdate

setdate(buffer)

d0	24	
d1	buffer	date/time buffer
Return	0 (-1 on error)	

Moves the date/time in the 7 byte area pointed to by buffer to the system time accumulator, provided the new date/time is acceptable. Returns 0 if the new date/time is acceptable, or -1 if an impossible date/time was supplied.

Get time/date string - gettdstr

gettdstr(buf, arg1, arg2, arg3)

d0	83	
d1	buf	string buffer
d2	arg1	varies
a0	arg2	
a1	arg3	
Return	buffer address, or various	

Arranges the current system date/time into the standard format:

HH:MM:SS DD MON YYYY

buf must point to a 21-byte area. The date/time string is null-terminated (ends with ASCII 0, or null, character.)

gettdstr`(0, 0, dateptr, mybuf)` will convert the 8 byte time pointed to by `arg2` into a human readable string form and place it at memory pointed to by `arg3`. In syscalls header as `cvttdstr`.

gettdstr`(0, 1, increment, x)` sets the date/time increment to `arg2`. In syscalls header as `settimeinc`.

gettdstr`(0, 1, 0, x)` returns the current value of the date/time increment. In syscalls header is `readtimeinc`.

The date/time increment is simply the number of microseconds between vertical sync interrupts. Normally 19968 (a nominal 50th of a second), this may be varied to trim the operation of the real time clock when different video modes are programmed.

Get ALT-C status - abortstat

abortstat( )

d0	25
Return	ALT-C status (0 for no ALT-C)

Try to avoid using this call. It is no longer used internally by the EPROMS. Use signals instead, as documented in the *Technical Reference Manual*. Signals generally are compatible with their use under Unix.

Returns the state of 1616/OS's abort flag. The flag is cleared by this call before it returns. The flag is set by an ALT-C only if the process is interactive. This means that a background task cannot be interrupted by Alt C.

The way to use this call is to use it once before you start polling ALT-C (at the start of your program, possibly) and discard the result; this clears the internal flag. From this point on, a call to '***abortstat***' returns true, if the user wants out.

Enable VIA timer1 interrupts - ent1ints

ent1ints(vec, preload)

d0	26	
d1	vec	pointer to user ISR code
d2	preload	ISR call period
Return	nil	

This call sets up the timer 1 output of the VIA to produce a stream of level 4 interrupts which vector to your code, which is pointed to by `vec`. This mechanism is used by the cassette write routines and the ***freetone*** system call. See ***set_vsec*** (syscall 16) for details of interrupt service routine (ISR) code.

The frequency at which your code is called is

750,000 / ((2 * preload) + 3.5) Hertz.

For speed purposes there is no intervention between 1616/OS and your code. The general format of your ISR should be:

```
V_AREG    equ         $700102
          org         wherever
my_isr    movem.l     <reglst>,-(a7) * Save registers
          bset        #2,V_AREG * Toggle /PRE
                                  * on U49 to clear
          bclr        #2,V_AREG * /CASIRQ.
          | |
          | |                     * Your code
          | |
          movem.l     (a7)+,<reglst> * Restore
          rte                     * Return from the ISR
```

The toggling of bit 2 of the VIA A port is essential. Don’t forget to disable VIA timer interrupts when you finish with your routine.

Disable VIA timer1 interrupts - dist1ints

dist1ints( )

d0	27
Return	nil

Disable (halt) timer 1 interrupts. This system call may be made from within an ISR.

Calculate a sine - sine

sine(angle)

d0	28	
d1	angle	Angle in the range 0 - 1023
Return	sine(2*pi * (angle/1024)) * 128	

This system call may be used for building look-up tables for sound generation (in association with the timer1 interrupts, see above). It may also be used to synthesise waveforms for use with the ***freetone*** system call (below).

The angle is taken, modulo 1024 and a value between 127 and -128 ($0000007f and $ffffff80) is returned. A few samples:

sine(0) = 0
sine(256) = 127
sine(512) = 0
sine(768) = -127

The signed 32-bit numbers thus generated may be multiplied by scaling factors and added together to produce musical waveforms. When the synthesised waveform table is complete the least significant bytes should be sent out through the DAC with bit 7 inverted. The inversion of bit 7 is needed because the peak excursions of the DAC are $00 and $ff, not $80 and $7f.

Define a function key - def_fk

def_fk(fknum, str)

d0	29	
d1	fknum	Number of function key - 1
d2	str	Pointer to definition
Return	0 (-1 if ‘fknum’ bad)	

Programs function key fknum+1 to produce the sequence of characters pointed to by str when it is typed. The string str is null-terminated; it is copied into 1616/OS’s data areas by this system call, so the original string need not be preserved. This is essentially identical to the fkey command detailed in the 1616/OS *Reference Manual*.

If fknum is in the range 64-73, then a pointer to the definition of function key fknum - 64 is returned. This means you can easily read back function key definitions (new in Version 4.0b).

Get random number seed - getrand

getrand( )

d0	30

During character input the system increments a 32 bit number. This system call returns the current setting of the number. This is useful for random number seeding and disk root block randomisation.

Request storage from system - getmem

getmem(nbytes, mode)

d0	62	
d1	nbytes	Number of bytes required
d2	mode	Storage mode
Return	Start address or negative error code	

There are seven supported modes to this system call, and they are 0 to 3, and 9 to 11. The informally documented ones were for test and development purposes (but since Andrew told me about them somewhere, I've included 4 to 8 - I don't know whether you can count on them in future releases).

Mode = 0

Allocate nbytes bytes of memory, return start address (an even address, to suit 68000 systems) or negative error code. The allocated memory is automatically freed upon termination of the ***exec*** system call which invoked the user program. This is the usual allocation mode. If nbytes is odd is it automatically incremented by one. If nbytes is zero it is set to 2.

Mode = 1

As with mode 0, except the memory remains allocated after the current ***exec*** terminates. This permits the permanent allocation of memory. Memory allocated with mode = 1 may be freed using the ***freemem*** system call. Mode 1 memory is for loading in programs which remain in memory, and for allocating storage from within an interrupt service routine (the memory allocator is re-entrant).

Mode = 2

The return value is the normally the size of the largest unallocated block of memory, divided by two. This has been done to confound programs (such as the HiTech C optimiser) which allocate all of the largest free block, which clogs the system. The nbytes argument is ignored.

The default divisor can be altered, using mode = 9. If the divisor is 1, this call is identical to its operation under Version 3.

Mode = 3

The return value is the total amount of free memory. The size of all the free blocks is summed and returned.

Mode = 4

Returns glbtable, a pointer to a bitmap. Bit 7 of byte 0 is set if $4000-$401f is allocated, bit 6 is $4020-$403f, etc.

Mode = 5

Endtable, a bitmap of the length of each run of glbtable and executables.

Mode = 6

glblongs, the number of longwords in glbtable and executables.

Mode = 7

Executables, a pointer to the start of a table of 64 pointers (one per process) to bitmaps of mode 0 memory use of all processes. Nil pointer to non-existant PID.

Mode = 8

Arenatop is the address of the highest location used by the memory manager.

Mode = 9

Memcheating. Contents of D1 (default value 2, normally containing nbytes) sets the divisor factor for mode 2. If mode 9 is used with D1 containing 1, then the next mode 2 will return the actual number of bytes in the largest free block.

Mode = 10

Returns the actual, real maximum free block size, irrespective of the divisor set by mode 9.

Mode = 11

Allocates a new PID for the `schedprep()` call (`syscall .131`), provided the nbytes argument is 0.

The memory allocation functions in 1616/OS generally print error messages out on standard error if something goes wrong. Memory allocation behaviour is now configurable on a per process basis, so that a process with memory allocation problems can be sent a sigsegv (signal 11) upon segmentation violation. See the ***oscontrol*** syscall, cmd 25, for details.

Allocate memory at address - getfmem

getfmem(addr, nbytes, mode)

d0	63	
d1	addr	Desired address
d2	nbytes	Amount of memory desired
a0	mode	Allocation mode
Return	addr (or negative error code)	

This call requests allocation of the nbytes of memory at address addr. If the memory is currently free, it is reserved and addr is returned. If some or all of the requested block is reserved, a negative error code is returned.

Main applications of this call are for loading in non-relocatable code (the system uses ***getfmem***( ) for `.exec` files), and for safely using multiple video pages. This syscall now fails **if** the requested address is not on a 32-byte boundary (the syscall formerly attempted to massage the address to a suitable value, causing problems).

The mode field is set to 0 or 1, and it has the same use as in the ***getmem*** system call.

Release allocated memory - freemem

freemem(addr)

d0	64	
d1	addr	Address of memory to release

Return	Block size, 0 or error code

Releases (for possible reuse) the block of allocated memory which starts at addr. Typically addr will have been obtained from a previous call to ***getmem*** or ***getfmem***.

If the most significant bit (bit 31) of addr is set, this system call returns the size of the block of memory at addr. This will be the same as the value of nbytes which was passed to ***getmem*** or ***getfmem*** when the block was allocated. The memory is not released if bit 31 of addr is set. Andrew often makes use of this ‘special effect if bit 31 set’ so watch out for it in your use of calls.

Alter/install a system call vector - setstvec

setstvec(vecnum, whereto)

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

This system call permits the alteration of how a specific system call is handled.

When a system call occurs, the system stacks d1, d2, a0, a1 and a2 and 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 whereto 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 whereto is negative the current setting is read from the table but no changes are made.

Line editor with length - nledit

nledit(str,len)

d0	84

d1	str	Pointer to string to be edited
d2	len	Maximum acceptable length of string

This is the same as the old ***ledit*** call except that the maximum string length is determined by len. The maximum value of len is constrained to 512.

This system call is designed for those instances where a fixed length record is required for storage in a database. Another use is where there are only a small number of screen columns available. len refers to the maximum number of characters, not to the maximum printable length of the string. If a typed string contains tabs or control characters, then its printed length may exceed its actual length. This means that a typed string could exceed desired bounds on the screen.

Line editor for dial up access - fnledit

fnledit(buf, len, in, out)

d0	142	
d1	buf	Pointer to space for edit buffer
d2	len	Maximum acceptable length of string
a0	infd	Input file handle
a1	outfd	Output file handle
Return	str	Pointer to start of string being edited

This call is for dial up access, so that some form of line editor is available for callers using really dumb terminals, Macintosh, IBM PC, etc., where they can’t supply internationally accepted Applix terminal control sequences.

It is a low level call to the 1616/OS line editor. Permits the source of input and output to be other than standard input and output. Do things like ead a line from a user without echoing to the screen (make the output file descriptor the NULL: device.

Some of you may wonder why this isn’t syscall .85. So did I. Syscall .85 was used for lbedit in the dark past, even Andrew dosn’t know what it did, and it should never again be disturbed. Syscall .85 is now badsyscall!

Line editor - ledit

ledit(str)

d0	86	
d1	str	Pointer to string to edit
Return	str	

Invokes the standard system line editor upon the null-terminated string at str. There must be at least 512 bytes spare at str. If you wish to use ***ledit*** for getting a new line from the user rather than editing an existing one, put a null at the start of the string.

The line editor returns zero if an end-of-file character (usually **^D**) is entered at the start of line. Otherwise it returns a pointer to the start of the line just edited.

Indefinitely call 1616/OS command executor - iexec

iexec(prompt)

d0	87	
d1	prompt	Pointer to prompt string or value
Return	nil	

Repetitively calls the line editor and command interpreter until the QUIT command (or end-of-file) is entered.

The argument prompt points to a null-terminated string which is used as the input prompt. When passed a a prompt string for display, the current directory name is substituted into a ‘%s’ in the path. For example, the system call

```
            iexec("[my name] %s")
```

will fire off an interactive shell with a prompt such as

```
            [my name] /f0/mydir>>
```

If this string is empty (i.e., prompt points to a zero byte) then the call returns after editing and executing just one command line. If prompt equals 1 (register d1 = $00000001) then the prompt is the current directory, provided this has been enabled with the OPTION command (option 0, refer *Users Manual*).

A number of ‘>’ characters equal to the depth of nesting of calls to ***iexec*** is displayed after the prompt.

Since Version 3.2b, **iexec** will return if an end-of-file character is typed in at the start of a command line. Thus, Ctrl D will act the same as QUIT (provided the EOF character is set to 4, instead of the default of 256 or no EOF). ***iexec*** will terminate when it receives a sighup signal.

From Version 4.0b, the value passed as the first argument (in d1) will alter the way ***iexec*** functions.

If prompt = 1, performs ***iexec*** until quit or EOF is passed. Prints an extra > in prompt if ***iexec***s are nested. Prompt contains current path if OPTION 0 is set.

If prompt = 2, performs ***iexec*** until quit, however only one > appears in prompt. Prompt contains current path if OPTION 0 is set.

If prompt = 3, performs ***iexec*** until quit, however only one > appears in prompt. Prompt does not contain current path.

If prompt = 4, performs ***iexec*** as a root shell (which can cause all sorts o software problems, since input may go to any root shell).

If prompt is 0, or a value from 5 to 32, the prompt essentially points to a null string, and ***iexec*** executes once. If it exceeds 32, then it is a pointer to a memory location, and will return the conetnts of that location as a prompt. The argument passed to prompt will appear as a value (at the end of the line) when you do a `ps`.

If bit 31 of prompt is set, the shell process which ***iexec*** sets up is run asynchronously. This means that the command

```
          syscall .87 80000002 < sa: >sa: }sa:
```

will start up an ***iexec*** (shell) process on serial channel A for a second user. This is one simple example of how to let a second (or third) user into the Applix 1616. A more convenient method is to use the `getty` program, available from Applix or the User Group.

Execute a 1616/OS command - exec

exec(str)

d0	88	
d1	str	Pointer to command
Return	0 (negative if error)	

Interprets a 1616/OS command. The null-terminated command string may invoke either inbuilt or transient commands. The command may include wildcards or I/O redirections.

The normal error messages are produced if any errors are detected.

If there is any form of error detected a negative value is returned (bit 31 of d0 set). Otherwise any non-negative value may be returned.

Before the ***exec*** completes it closes and restores standard input, output and error. It then closes any files which were opened by this ***exec*** and not closed (for Version 3, if enabled - OPTION 7, *Users Manual* - this option is changed in Version 4). All memory which was allocated by this ***exec*** is deallocated unless it was allocated with mode set to 1 - see the *Technical Reference* manual for details.

Call a memory resident driver - callmrd

callmrd(mrdno, cmd, arg)

d0	89	
d1	mrdno	Driver number or name
d2	cmd	Command to driver

a0	arg	Argument passed to driver.
Return	Value from MRD, -1 if bad mrdno	

Calls memory resident number mrdno by stacking arg at 8(sp) and cmd at 4(sp) and calling the MRD's start address. See the MRD documentation in the *Technical Reference Manual* for more details.

If mrdno = -1 then the return value is a pointer to the following internal data structure. Note that a lot of the fields in this structure are read from the `mrdrivers` file at boot time. If no `mrdrivers` file was found they default, as described in the MR drivers documentation.

long	magic1	Magic number, always $601a which is actually BRA.S *+28
long	vers	1616/OS version of MRDRIVERS
long	rdsize	Size of RAM disk in kbytes 200 default
long	memusage	Total length of all MRDs in memory
long	ndrivers	Number of MRDs in memory
long	magic2	2nd magic number, $d80ab7f1
long	stackspace	Size of system stack $10000 default
long	vramspace	Size of VRAM space $8000 default
long	obramstart	Start address of the 512k of on-board RAM
long	mrdstart	Address of the header of the first MRD in memory
long	rdstart	Address of block 0 of the RAM disk
long	bitmaps	Address of the 7k of buffers for the bitmaps of /RD, /F0, /F1, /H0 /H1
long	vramstart	Address of the start of video RAM
long	colours	Screen colours
short	nlastlines	Last line recall depth line editor
short	ncacheblocks	
ushort	maxfiles	Max number of file control blocks
ushort	ndcentries	Number of dir cache entries
long	chardrivers	Pointer to 16 chardriver structs
	{28 shorts, for future expansion}	

Alter keyboard scan code vector - set_kvec

set_kvec(vec)

d0	90	
d1	vec	Pointer to key scan interpreter.
Return	Previous vector	

The 1616 keyboard is interrupt driven. Each time the keyboard transmits a scan code an interrupt is generated and an ISR is called. You may supply your own ISR to interpret keyboard commands using this system call. vec points to your code.

Each time a key is pressed or released the keyscan code is pushed onto the stack, accessible at 4(a7) and your code is called. See the IBM keyboard documentation for the keyboard scan codes. Note that the existing keyboard driver (from V3.2b on) ignores $e0 and $e1 characters from keyboards with strange extensions.

If this system call is performed with a vec argument of 0 then the default 1616/OS scan code interpreter is installed. This interpreter is reinstalled at all levels of reset.

If vec is 1, the keyboard scan is set to raw mode. All scan code interpretation is turned off, so calls to ***getchar()*** or ***read()***, etc will return raw key codes. This will really stuff up programs like `vcon` or `mgr` that expect ASCII codes. If you do use this raw mode, set the `con:` device into raw mode also, to avoid problems with codes corresponding to xon, xoff, eof, reset, etc.

If vec is 2, clear keyboard raw mode state, return to previous state.

If vec is 3, read the current raw mode state.

This system call returns the previous contents of the keyscan interrupt vector, so your code can pass the scan code on to the old interpreter if desired. This discipline permits any number of memory resident programs to inspect the keyboard stream. This previous vector is tricky. It points to a pointer to the code, not directly to the code.

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. See the *Technical Reference Manual* for additional details. See also the description of command line transient arguments in Chapter One of this manual.

Sort things in memory - qsort

qsort(base, nel, width, compar)

d0	92	
d1	base	Table start address
d2	nel	Number of elements in table
a0	width	Size of each element in bytes
a1	compar	Pointer to comparison function
Return	nil	

This is a general implementation of the non-recursive Quicksort algoritm. The algorithm sorts an array of elements pointed to by base. This is a very general function which may be used to sort practically any form of data.

nel is set to the number of elements in the array.

width is the size in bytes of each element in the array.

compar is the address of a user-supplied comparison function. The function is called by ***qsort*** to determine which of two elements is considered the 'least' for sorting purposes. The user-supplied comparison function is passed two pointers at 4(sp) and 8(sp). It must return a number less than zero (bit 31 of d0 set) if the element pointed to by 4(sp) is considered to be less than that pointed to by 8(sp). It must return 0 (zero) if the two elements are equal. It must return a number greater than zero (bit 31 of d0 clear) if the element pointed to by 8(sp) is the least of the two.

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
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. See the full description in the *Technical Reference Manual.*

Find CPU clock speed - cpuspeed

cpuspeed

d0	94
Return	0 for 7.5 MHz 1 for 15 MHz

If the system clock is running at 7.5 MHz, this call returns zero. If the system is running at 15 MHz, then a 1 should be returned. Unfortunately, this got changed to an 8 long ago, and Andrew can’t recall why (except he does say he had a good reason).

Execute argument array - execa

execa(argv)

d0	97	
d1	argv	Pointer to pointers to arguments
Return	Return value from the execution of the command	

This is a lower-level way of executing a system command. The register d1 contains a pointer to a table of pointers to null-terminated strings. The first string is the name of the command (inbuilt command, transient program or executable memory resident driver) to be executed. The second string is the first argument to be passed to the command, etc. The table of pointers is terminated by a zero (nil) pointer. Wildcards in the passed strings are not expanded. Only one command may be executed (the "!" command separator is not understood at this level).

The ***execa*** system call is the lowest level ***exec*** command. ***iexec***, ***exec*** and ***execv*** all call ***execa*** to do the real work. See the section on Multitasking in the *Technical Reference Manual*, for details of the ***aexeca*** asynchronous and upgraded version of this call.

Example: A program fragment which uses the 1616 assembler to assemble the file ‘test.s’.

```
          -------
          move.l     #asmname,argvbuf
          move.l     #testname,argvbuf+4
          clr.l      argvbuf+8 * Terminal nil pointer
          move.l     #argvbuf,d1
          move.l     #97,d0
          trap       #7          * Do the assembly
          tst.l      d0          * Error?
          bmi        errorhandler
          -------
asmname   dc.b       'SSASM',0
testname  dc.b       'TEST.S',0
argbuf    ds.l       10          * Pointer array
```

Execute command with arguments - execv

execv(path, args)

d0	98	
d1	path	Pointer to null-terminated command name
d2	args	Pointer to table of pointers to arguments
Return	Return value from command execution	

This is very similar to the ***execa*** call. path points to the command or program to be run, args points to a table of pointers to null-terminated strings which become the arguments to the command at path when it is run. Again, the args array is terminated by a nil pointer after the last argument pointer.

Set/read a system option setting - option

option(opnum, setting)

d0	99	
d1	opnum	Option number
d2	setting	Option setting
Return	Old option setting	

This system call is very similar to the 1616/OS inbuilt command OPTION. The arguments opnum and setting behave in the same manner as the options to the OPTION command.

This call returns the previous setting of the selected option.

If bit 31 of opnum is set then the current setting is read, but no change is made.

If opnum is outside the valid range of options, a negative error code is returned.

Error number interpreter - errmes

errmes(ec)

d0	122	
d1	ec	1616/OS error number
Return	Pointer to human-readable string	

This system call is similar to the ***interpbec*** call (syscall 114). It takes a 1616/OS error number (negative) and returns a pointer to a null-terminated string (within the ROMs) which is the appropriate human-readable message. If ec is not a known error message, the string ‘Unknown error’ is returned. If ec is not negative the string ‘No error’ is returned. The ***printf***, ***fprintf***, and ***sprintf*** system calls can now directly produce error messages using the #e output variable, so you do not need to use ***errmes*** as often.

Return 1616/OS version - getromver

getromver( )

d0	126
Return	1616/OS version number

Returns a byte in d0 which indicates the version number of the ROM. The operating system versions will be of the form X.Y, where X and Y are decimal numbers between 0 and 15. On return from this system call bits 0-3 of d0 contain Y and bits 4-7 contain X.

You should use this system call to determine whether your transient program is running under a suitable version of 1616/OS.

4
File and Block I/O Calls

The file I/O system calls allow transient programs to manipulate disk files and character devices.

Introduction

The file I/O system is designed to work transparently upon disk files and character devices (such as SA: and CENT:). This means that character devices can be opened, closed, read from, written to and so on. For this reason we introduce the term *stream*. A stream is a source of input and/or destination of output which may be an open file or a character device.

Each stream is identified by a handle, which is a number in the range 0 to 31. Character devices have handles in the range 0 to 15; open files have handles in the range 16 to 31. Handles are used to reference a stream for I/O operations and manipulations. The character devices are con: (0), sa: (1), sb: (2), cent: (3) and null: (4).

Earlier versions of 1616/OS required 16 to be added to file handles before they could be used in the ***getc***, ***putc***, ***sgetc*** and ***sputc*** system calls. This is no longer necessary, and the system now subtracts 16 from file descriptors which are in the range 32-47, so programs written under 1616/OS version 2 which use this kludge will still work.

The file I/O system calls work transparently when the name of a character device (ending in a colon) is used in place of a file pathname.

All of the disk/block device system calls return error codes in d0 if an error is detected. An error code is a negative 32 bit number (bit 31 set). There are a range of error codes implemented; call using ***interpbec*** or ***errmes*** system calls, or the %e option in ***printf*** and similar routines. The errors are listed in Appendix B, and can be displayed with syscall .122, .114 or the %e option on the ***printf*** and similar print routines.

A maximum of about 230 files may be open simultaneously. Each file redirection uses one of the sixteen file control blocks. Each currently open shell file uses one also, so don’t open too many windows.

Files currently have no size limit. The previous 512k limit has been avoided by ensuring that the blockmap for a file extends across contiguous blocks, one block per half megabyte. This means that every time the system grows a file across a 0.5 Mbyte multiple, it **must** relocate the entire blockmap. If there is nowhere to put the blockmap, a "disk full" error results. Incidently, older versions of `fscheck`, the file system checking program, do not understand files longer than 0.5 Mbyte. Be warned.

The ‘working position’ in a file, or the ‘current file pointer’ refers to the position within the file (relative to the start) where the next read/write will occur. The pointer is automatically advanced by reads and writes. The first byte in the file is positioned zero bytes relative to the start, so to read it would require a working position of zero. There is a nasty, but rare, ‘gotcha’ associated with this, but I’ve forgotten the details.

File control block

Each file has a file control block structure associated with it. This consists of the following:

file_name	char	The full pathname
openmode	ushort	Mode in which file was opened
bdvrnum	ushort	Appropriate driver number
blkmap	ushort	Pointer to file’s block map
bmindex	ushort	Current index into blkmap[]
blkbuf	char	Pointer to buffer for disk I/O
dirblk	ushort	File’s directory entry block
dirindex	ushort	Position in directory block
file_size	uint	Accumulated length
file_pos	uint	Where we are in it
blkmapblk	ushort	Where file’s block map starts on disk
changed	ushort	Flag. Current block must be written out
blkstat	ushort	Flag 0. Block must be read
nusers	ushort	Number of processes using file
owners[maxpids]	char	PIDs who own or have inherited it
pipefcb	struct	If non-zero, it is a pipe
uid	ushort	UID the file will get
needsflush	ushort	Altered since last close (80000000 | fd)

Standard input, output and error

When a program is running, it does not know (or need to know), which physical device its output is actually going to. The output of the program is instead sent to ‘standard output’, which is an output stream which the operating system manages. The handle for standard output is always $101 (257 decimal) or $81, regardless of what physical device is currently handling output. The user may redirect the physical device to which the operating system sends standard output with redirection commands (`>filename` or `>device:`).

Similarly, standard input is a character stream which the operating system translates into the currently assigned input device. The handle for standard input is always $102 (258 decimal) or $82, regardless of the currently assigned source of input. The user may vary a program’s source of standard input, for the duration of the program’s execution, by using the ‘`<filename`’ or ‘`<device:`’ redirections when invoking the program. As an alternative, you can alter the standard input or output devices with the ***set_sip***, and ***set_sop*** system calls.

As an added convenience, a third stream, 'standard error', is implemented. This stream has a handle of $100 (256 decimal) or $80. This stream is provided for outputting error messages. The user can redirect the destination of standard error using the '`}filename`' and '`}device:`' redirections. This permits the separation of a program's normal output from its error messages from the command line. This can also be redirected by using the ***set_ser*** system call.

Either value ($10x or $8x) will work. The versions with the eighth bit set were added because HiTech C does not handle file descriptors exceeding $ff with any grace (it expects uchar).

File system interlock

Much of 1616/OS is now re-entrant to support multitasking, so a routine within the EPROMS can be used simultaneously by two or more processes without becoming confused. The exception to this is the file system. Since it may be impossible to write a re-entrant file system, 1616/OS Version 4 uses an interlock which ensures that only one process at a time is within the file system.

When a process starts to use the file system, it is 'locked in' until it reaches the stage of actually performing physical I/O. At this time, the lock is released, and other processes may be scheduled. If any of these processes attempt to use the file system, they are put to sleep until the process which is currently using the file system has finished its I/O, and terminated whichever file system call it was using.

The net effect of this is to serialise access to the file system: only one process can use it at a time. Access is granted on a first-come, first-served basis. There is minimal interference to processes (such as `edit`) which have little use of the file system.

If a file is being written-to (say as a log file) by a background process, it is possible for another process to read from the same file. The ***open*** system call has provision for this.

File and block I/O calls

Set out below are all the actual system calls for the file system. As you would expect, many correspond rather directly with the normal command line operations with which you should already be familiar.

Change current directory - chdir

chdir(path)

d0	65	
d1	path	Pointer to new pathname
Return	Error code or current path	

This is very similar to the CD inbuilt command. path may be a relative pathname. The system changes to the new directory and reads it into the current directory cache. The return value is zero or a negative error code.

If path is zero (a nil pointer) then the current directory is not changed and a pointer to the full pathname of the current directory is returned. Use this if a process needs to change its own current directory; do not allow a process to use ***proccntl*** cwd upon itself.

Make a new directory - mkdir

mkdir(path, ndirblks)

d0	66	
d1	path	pathname or directory to create
d2	ndirblks	
Return	Error code or 0	

Creates a new directory as specified by path, which may be an absolute or relative pathname. ndirblks is the number of 1024 byte blocks which the directory is to occupy. Note that this differs from the size specification in earlier versions of the mkdir inbuilt command.

The system searches for ndirblks contiguous free blocks on the selected device and locates the directory there. It is for this reason possible that a directory may not fit on a nearly full, fragmented disk which would appear to have room for it.

If the free blocks are found they are reserved, the directory is initialised and the links to it from its parent directory are created.

Expand out a pathname - getfullpath

getfullpath(path, memmode)

d0	67	
d1	path	Pointer to pathname
d2	memmode	Memory allocation mode
Return	Pointer to full path	

This system call takes the full or relative pathname at path, converts it into a full path, and returns a pointer to the full path. The memory in which to store the resulting full pathname is obtained from ***getmem***. The memmode argument here is passed on to ***getmem***, hence it defines whether or not the memory in which ***getfullpath*** places its output is released on termination of the currently running program.

Your program should free the memory which was allocated by ***getfullpath*** when it has finished with it. This is done by passing the address which ***getfullpath*** returned on to ***freemem***. This is not required as often, with removal of the old `volumes` facility.

Compare pathnames - pathcmp

pathcmp(path1, path2)

d0	68	
d1	path1	Pointer to null-terminated pathname
d2	path2	Pointer to null-terminated pathname
Return	Zero if same file	

Under 1616/OS block devices may be referred to by their physical identifier, such as /F0. The ***pathcmp*** system call gives access to an internal function which compares two 1616/OS pathnames and returns zero if they refer to the same file. The filenames may be relative to the current directory, or absolute.

Install a block device driver - inst_bdvr

inst_bdvr(br, bw, bs, name, pv)

d0	100	
d1	br	Block read entry point
d2	bw	Block write entry point
a0	bs	Block device status entry point
a1	name	Pointer to name of block driver
a2	pv	Pointer to bit map buffer
Return	Driver number (negative if installation error)	

See the *Technical Reference Manual* for details.

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	

See the *Technical Reference Manual* for details.

Raw block read - blkread

blkread(blk, buf, dev)

d0	103	
d1	blk	Number of block to be read
d2	buf	Memory address for transfer
a0	dev	Block device driver index
Return	0 (or error code)	

This call reads the selected 1024 byte block from the nominated device driver to main memory starting at buf. If an error occurs the error code is returned. The device driver numbers are /RD (0), /F0 (1), /F1 (2), /H0 (3), /H1 (4). Between this and the monitor commands, you can do a fair job of investigating most low level file characteristics.

Raw block write - blkwrite

blkwrite(blk, buf, dev)

d0	104	
d1	blk	Number of block to be written
d2	buf	Memory address for transfer
a0	dev	Block device driver index
Return	0 (or error code)	

This call writes a 1024 byte block from memory at buf onto the selected block on the nominated device driver. If an error occurs, the error code is returned.

Create for output - creat

creat(pathname, stat, addr)

d0	108	
d1	pathname	Pointer to null-terminated pathname
d2	type	Status bits
a0	addr	File load address
Return	Stream handle	

Creates a new file, the name of which is pointed to by pathname. The stat or type field sets the file's attribute bits, as described in the documentation for the 1616/OS `filemode` command. The load address addr is that at which the program is executable; transient `.exec` programs load at this address. Make the load address field zero for files which are not `.exec` files.

If pathname points to an identifier for a character device (the last character being a colon) this system call returns the handle of that device. This may be treated in the same manner as a file handle, so character devices and files may be treated identically.

If the named file already exists it is deleted by this command, together with its status bits.

The file is prepared for output. ***Seek***s and ***read***s are permitted.

The command returns a file or character device handle.

Prepare for input - open

open(pathname, mode)

d0	105	
d1	pathname	Pointer to null-terminated pathname
d2	mode	Open mode
Return	handle (or error code)	

Prepares a file or character device for I/O, depending upon mode.

mode = 1 o_rdonly.
Read only. Writes fail. File pointer points to start of file.

mode = 2 o_wronly.

mode = 2 o_append (formerly named o_wronly)
Append, read and write, same as mode 2. ***Write***s and ***seek***s allowed. Files open for writing are always also open for reading. Note that the file pointer points to the **end** of file.

mode = 3 o_rdwr
Read and write. File pointer points to start of file.

mode = 4 o_truncate
Truncate a file. File size if reduced to 0, opened in R/W mode, with file pointer at start (which is also end of file). This mode was added in Version 4.2 because ***creat()*** needed to use it when a file being created already exists. Makes ***creat*** more efficient, and simplifies the preservation of the file attributes and permissions.

You can open for reading a file that is already currently open for writing. This permits the inspection of files which are being written to by ongoing background processes.

If the ***open*** fails then an error code which is negative (bit 31 set) is returned. Use ***errmes*** or ***interpbec*** to interpret the code.

Successful ***open***s return a handle (a 32 bit integer) which is used for identifying the output stream for the rest of the time that it is open.

If pathname refers to a character device then a handle is returned which may be used to read and write that device.

The o_truncate was added because there was no convenient way to return permission bits of a file otherwise. There should be a file `openmode.d` that discusses this.

Read from a stream - read

read(handle, buf, nbytes)

d0	106	
d1	handle	file descriptor from ***open***
d2	buf	pointer to transfer area
a0	nbytes	number of bytes to read
Return	Number of bytes read or error code	

Reads nbytes bytes from the file or character device described by handle to memory at buf. The call attempts to transfer nbytes bytes, however if there are less than this number of bytes left in the file or if an end-of-file character is received by a character device then a smaller number will be transferred. A zero is returned at the end of a file. The file pointer is advanced by the number of bytes read. If an error is detected a negative error code is returned. See the ***set_kvec()*** syscall if you require raw scan codes input.

Write to a stream - write

write(handle, buf, nbytes)

d0	109	
d1	handle	stream handle from ***open/creat***
d2	buf	pointer to transfer area
a0	nbytes	number of bytes to write
Return	0 (negative code on error)	

Writes nbytes bytes from memory at buf to the file or character device described by handle. The file pointer is advanced by nbytes. A negative return is an error code.

Close a disk file - close

close(handle)

d0	107	
d1	handle	file handle from ***open/creat***
Return	0 (negative code on error)	

Flushes buffers if the file described by handle is open for output. Frees file buffers. Returns negative value on error.

If handle refers to a character device no action occurs and a zero is returned.

If handle is -1, it syncs the file system. All files open for writing are flushed out, but remain open so that concurrent reads may continue.

If the file descriptor handle has its top bit (bit 31) set to 1, the file associated with the descriptor handle is flushed to disk. The directory entry is updated, so that other processes can open and read the file. It remains open for writing. The blockmap is not written out when closing files of zero length.

This action is provided in Version 4 so that background processes that log to a file can flush the file each time they write something to it. All the process need do is ***close***(handle | 0x80000000) each time something is written.

If handle is made greater than 127, and the top bit is not set, ***close*** returns zero. This prevents any attempt to close STDOUT, STDERR or STDIN, ensuring they will stay open across ***exec***s.

Delete a disk file or directory - unlink

unlink(pathname)

d0	110	
d1	pathname	pointer to filename
Return	0 (negative code on error)	

Deletes the file or directory whose null-terminated name is pointed to by pathname. If the reference is to a directory it is only successfully deleted if the directory is empty and does not lie in the path of the current directory.

Rename a disk file - rename

rename(oldpath, newname)

d0	111	
d1	oldpath	pointer to current pathname
d2	newname	pointer to desired filename

Return	0 (negative code on error)

Changes the name of the file or directory whose null-terminated pathname is pointed to by oldpath to that pointed to by newname.

Get the status of a disk file/directory - filestat

filestat(pathname, buf)

d0	112	
d1	pathname	pointer to file/directory pathname
d2	buf	pointer to 64 byte buffer area
Return	varies	

This system call scans the disk directory and the file control blocks for the entry of the file/directory whose name is pointed to by pathname. If found the directory entry is read to the area pointed to by buf and a code is returned.

If passed a number in the range 0-31, in place of `pathname` (d1), it returns a pointer to an output driver (if in the range 0-15), or to a file control block (if in the range 16-31). These numbers correspond to the handles returned from the ***open*** and ***creat*** system calls. This can be used to obtain the name of a file or character device from its handle.

Unfortunately, for obscure historical reasons the two are different. The character device driver consists of three longwords (12 bytes) followed by a 16 char null-terminated name. The data file structure is a POINTER to the null-terminated absolute (complete) pathname of the file (see the header file `files.h` for the gory details).

The structure of a directory entry is described in the *Technical Reference Manual*.

Return codes

0	File present, closed
1	File present, currently open for reading (mode 1)
2	File present, currently open for writing (mode 2)
other	Negative error code (-17 for file not there), or may return modes 3 or 4 for some file modes.

Sequentially read disk directory - readdir

readdir(dev, buf, dp, pos, pd)

d0	113	
d1	dev	block driver number (obtained from ***find_bdvr***)

d2	buf	pointer to block work area
a0	dp	pointer to 64 byte directory entry
a1	pos	position within directory
a2	pd	pointer to parent directory entry
Return	new position (or negative error code)	

This call permits sequential disk directory scanning. dev is the block device driver number returned from the ***find_bdvr*** system call; buf is a pointer to a 1024 byte block buffer which must remain unaltered throughout the directory scanning; dp is a pointer to a 64 byte directory buffer area; pos is the last value returned by ***readdir***, or 0 on the first call.

A new addition to this call is pd, a pointer to the parent directory entry for this directory. This must be provided for ***processdir*** to find the desired directory. The parent directory entry may be read into memory with a ***filestat*** on the directory pathname.

To use this call, set up the data structures and call ***readdir*** with pos = 0. The call returns a number which must be saved and used as pos on the next call to ***readdir***. Eventually -1 is returned at the end of the directory. Each call reads a directory entry to the memory pointed to by dp. Unused (free) directory entries (first byte of filename = 0) are skipped.

An easier method may be to use the ***rdalldir*** syscall to get a complete directory.

Read all directory entries - rdalldir

rdalldir (path, memmode, sortmode, psize)

d0	124	
d1	path	pointer to null teminated pathname
d2	memmode	memory mode, as per ***getmem***
a0	sortmode	directory sort mode
a1	psize	pointer to longword
Return		pointer to memory buffer, or error

Reads a complete directory into memory, for further manipulation. Enter the call with a d1 containing a pointer to a null terminated string containing the pathname. The memory mode (d2) is passed as an argument to ***getmem***, and would normally be 0 for memory that is not retained, and 1 for memory to be retained after termination of the current ***exec***. The directory sort mode (a0) is 0 for no sorting, 1 for sorted by date, 2 for sorted alphabetically by file name, same as in `option 2`.

The psize argument in a1 points to a longword within the calling program's data areas. This longword is altered by the ***rdalldir*** syscall to reflect the number of 64 byte directory entries which were read from disk. That is, the amount of memory allocated is 64 times the contents of the longword pointed to by psize. If you get an error code returned, then this result is meaningless.

The call returns a pointer to the memory allocated for the complete directory. You can inspect it with the memory manipulation commands, or use it in your programs. This call even reads empty entries, so your code should check for and skip over empty directory entries. An empty entry has a zero as the first byte of the file name. The caller must free this memory after use; simply pass the address to ***freemem***.

Interpret a block device error code - interpbec

interpbec (ec, buf)

d0	114	
d1	ec	error code
d2	buf	pointer to 50 byte buffer
Return	buf	

This system call interprets the block and file error codes. ec is the negative error code produced by a previous file/block call. buf points to a 50-byte buffer where a null-terminated error message string for the particular error code is assembled.

The easiest way to produce readable error messages is to use the %e parameter on the ***printf*** syscall, which will take an error number and print out the appropriate error message. See also ***errmes***.

The error codes implemented in 1616/OS are listed in Appendix B.

Seek to a new disk file position - seek

seek(handle, offset, mode)

d0	115	
d1	handle	file handle
d2	offset	where to seek to
a0	mode	seek mode
Return	new file pointer position (or negative error code)	

Moves the file pointer to a different position. This may be done on files which are open for reading only (mode 1) and those which are open for reading and writing (listed as mode 2, but I thought it was mode 3).

A ***seek*** on a character device returns zero.

There are three ***seek*** modes:

mode = 0
Absolute seek. Treat offset as an absolute position and seek to it.

mode = 1
Relative seek. Seek to current position + offset; may seek forwards or backwards (offset negative).

mode = 2
Seek to end of file (for appending)
(On a clear disk you can seek forever).

The results of this call may now differ, since there now appear to be additional modes to ***open***.

Return current disk file position - tell

tell(handle)

d0	116	
d1	handle	file handle
Return	current file position pointer (or negative error code)	

Gives the current working position in the file, relative to the start. Returns zero if handle refers to a character 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 information
Return	result of call	

This system call is documented in the *Technical Reference Manual*.

Manipulate directories - processdir

processdir(pathname, buf, mode)

d0	118	
d1	pathname	Pointer to name of file/directory
d2	buf	Various, mainly as pointer

a0	mode	Process mode
Return	Negative error code or directory position.	

The low level internal directory manipulation function. Avoid using, where possible. Documented in *Technical Reference Manual*.

Multiblock I/O - multiblkio

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

d0	119	
d1	drv	Block device number
d2	cmd	Function
a0	addr	Read/write address
a1	blockspec	Start block number OR pointer to block list
a2	nblocks	Number of blocks to read/write

See *Technical Reference Manual* for full details.

Symbolic links

Reserved for Jeremy Fitzhardinge's symbolic links code.

d0	134

This will be documented in the *Technical Reference Manual*.

Check permissions - chkperm

chkperm(pdirent, mask, fullpath)

d0	141	
d1	pdirent	Pointer to directory entry
d2	mask	Access mode
a0	fullpath	name of file being accessed
Return		bec_noperm or 0

This system call 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 upon the full pathname. See the *Technical Reference Manual* for full details.

5
Character I/O Calls

Introduction

1616/OS V4.0 supports up to 16 character device drivers. As mentioned in the introduction to section 4 of the *Programmer's Manual*, character devices may be operated upon as if they are files. Conversely, files may be operated upon as if they are character devices which supply one character at a time.

Again, character devices and files are streams which are represented by a stream handle, a number in the range 0 to 15 for character devices and between 16 and 31 for open files.

The special handles for standard input ($102 or $82), standard output ($101 or $81) and standard error ($100 or $80) may be used with the character I/O system calls. The $8x values were added because HiTech C does not expect a 9th bit here.

The end-of-file character formerly had no effect upon the character I/O system calls (particularly ***getc***). All characters came through literally, and user programs had to check for end-of-file if desired. Since EOF char support was added (V3.2), ***getchar*** and ***getc*** return a value of -18 (read past EOF) when an end-of-file character is typed in.

Character device drivers are cleared and reinstalled upon all levels of reset, however the EOF character is preserved on the CON: device.

Read one character from standard input - getchar

getchar( )

d0	2
Return	character (or negative error code)

This is the standard character input entry point. It supports all redirection (device file) so that transient programs which use ***getchar*** for user input will support all the standard 1616/OS redirection features.

A long word is returned. If it is negative then it is an error code from a file I/O error encountered during file I/O redirection.

If during file redirection, an end-of-file is detected on the input source, then the read-past-eof error code (-18) is returned. On the next call to ***getchar*** the source of standard input reverts to that which prevailed before the redirection (probably the keyboard). See ***set_kvec()*** syscall for details of keyboard raw input modes.

Get status of standard input device - sgetchar

sgetchar( )

d0	3	
Return	status	

Returns the status of the standard input device. If any characters have been received from the device this call returns non-zero.

If standard input is a file redirection the a value of 1 is returned.

If standard input is a device then the number of characters received into that device's input buffer is returned.

For generality it is best to simply test the return value for equality with zero, rather than using the number of buffered chars.

Put a character to standard output - putchar

putchar(ch)

d0	4	
d1	ch	character for output
Return	0 (or error code)	

Sends a character to standard output, supports all redirection, ALT-S command. Returns a negative error code if something goes wrong, particularly if output is a file redirection.

Get status of standard output device - sputchar

sputchar( )

d0	5	
Return	output device status	

If the output is a character device, this call returns the number of bytes which can be sent before the device's output buffer fills. If output is a file, 1 is returned.

It is best to simply test the result of this call for equality with zero.

Get a character from a stream - getc

getc(handle)

d0	6	
d1	handle	character device or file handle

Return	character or negative error code.

Gets a character from the specified stream. handle is the return value from an ***open***, ***creat*** or ***find_driver*** system call. It may refer to a file or to a character device.

If an error is detected during input (particularly from a file) a negative code is returned. Returns a value of -18 (read past EOF) when an end-of-file character is typed in.

If handle equals $102 or $82 the character is read from the current source of standard input, like ***getchar***.

Get status of an input stream - sgetc

sgetc(handle)

d0	7	
d1	handle	character device or file handle
Return	status	

Returns the status of the input stream (character or file) identified by handle. If handle equals $102 or $82 the status of standard input is returned. See ***sgetchar*** for details about the return value.

Put a character to an output stream - putc

putc(handle, ch)

d0	8	
d1	handle	character device or file handle
d2	ch	character to send
Return	error code (if file stream)	

Puts a character out to the stream identified by handle. For file output an error code may be returned.

Get status of an output stream - sputc

sputc(handle)

d0	9	
d1	handle	character device or file handle
Return	status of output stream	

Returns non-zero if the stream identified by handle can accept at least one more character. If handle refers to a character device then zero is returned if there is no more room in the output queue. If handle refers to a file then a value of 1 is returned.

Assign standard input - set_sip

set_sip(handle)

d0	14	
d1	handle	character driver or file handle
Return	previous standard input stream handle	

Sets the source of standard input to the stream identified by handle. This may be a file or character device. This is a permanent change. All calls to ***getchar*** or ***sgetchar*** or ***getc***($102 or $82) or ***sgetc***($102 or $82) are affected by this.

The current setting of standard input may be read by passing handle equal to -1. The setting of standard input is unchanged in this case.

Assign standard output - set_sop

set_sop(handle)

d0	15	
d1	handle	character device or file handle
Return	previous standard output stream handle	

Sets the destination of standard output to the stream identified by handle. This may be a file or character device. This is a permanent change. All calls to ***putchar*** or ***sputchar*** or ***putc***($101 or $81) or ***sputc***($101 or $81) are affected by this.

The current setting of standard output may be read by passing handle equal to -1. The setting of standard output is unchanged in this case.

Assign standard error - set_ser

set_ser(handle)

d0	20	
d1	handle	character device or file handle
Return	previous standard error stream handle	

Sets the destination of standard error to the stream identified by handle. This may be a file or character device. This is a permanent change. All calls to ***putc***($100 or $80) or ***sputc***($100 or $80) are affected by this.

The current setting of standard output may be read by passing handle equal to -1. The setting of standard output is unchanged in this case.

The ***set_sip***, ***set_sop*** and ***set_ser*** system calls will accept the special handles for standard input, output and error ($102 or $82, $101 or $81 and $100 or $80). For example, ***set_sop***($100 or $80) will set standard output to the same device which is currently handling standard error.

Locate a character device driver - find_driver

find_driver(ioro, name)

d0	95	
d1	ioro	not used
d2	name	pointer to null terminated device driver name
Return		device handle, or pointer to chardriver structure or error code

If name is less than 16, 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. This syscall was extensively reworked as at Version 4.2a.

The following device drivers are installed in the 1616 at power-on time:

Input drivers:

Device name	Character device number	Physical port
CON:	0	1616 keyboard
SA:	1	Serial channel A
SB:	2	Serial channel B
NULL:	4	Null device (throw away)
TTY:		A pseudo device

Output drivers:

Device name	Character device number	Physical port
CON:	0	1616 video display
SA:	1	Serial channel A
SB:	2	Serial channel B
CENT:	3	Parallel printer
NULL:	4	Bit Bucket

Install an input character device driver - add_ipdvr

add_ipdvr(io, stat, name, pv)

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 identified code
a1	passval	Value passed to input code and status code
Returns	Character driver number or -1 if no room	

The **add_ipdvr** and **add_opdvr** system calls are used to install character device drivers into the system. Refer to the *Technical Reference Manual.*

Install an extended input character device driver - add_xipdvr

add_xipdvr(iovec, stat, name, passval, miscvec)

d0	10	
d1	iovec	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 identified code
a1	passval	Value passed to input code and status code
a2	miscvec	
Returns	Character driver number or -1 if no room	

The **add_ipdvr** and **add_opdvr** system calls are used to install character device drivers into the system. Refer to the *Technical Reference Manual.*

Install an output character device driver - add_opdvr

add_opdvr(io, stat, name, pv)

d0	12	
d1	io	Pointer to character output code
d2	stat	Pointer to character output status code

a0	name	Pointer to colon null terminated device identified code
a1	passval	Value passed to output code and status code
Returns	Character driver number or -1 if no room	

If an install is done with `io` set to 0, the named driver is deleted correctly. Character devices are cleared and reinstalled at all levels of reset. The EOF character is preserved in `con:`.

Locate the character device driver table - get_dvrlist

get_dvrlist(ioro)

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

Each character device driver (input or output) which is installed in the system is identified by the following 28 byte data structure:

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

The writing and installation of character device drivers is described in section 4 of the *Technical Reference Manual*.

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. This call keeps track of and allocates the character device buffers.

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 ***exit***ed, 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 within the $400 to $3c00 space. Do this before returning to 1616/OS if the buffers are only temporary.

addr = 1 Set the buffer size to len bytes, allocated as mode 1 memory. This will be freed by the system when the buffer size is again altered. Great for programs needing a temporary larger buffer.

addr = -1 Return a pointer to the circular buffer structure associated with device dev. This is described in header file `storedef.h`.

Formatted output - printf

printf(contstr, p1, p2, p3, p4)

d0	48	
d1	contstr	format control string
d2	p1	first print argument
a0	p2	second print argument
a1	p3	third print argument
a2	p4	fourth print argument

Return	nil

This is an implementation of the C language ‘printf’ function. Refer to one of the many ‘C’ programming manuals for a comprehensible description of this function.

contstr is a pointer to the null-terminated format control string. The characters in this string are printed out including control characters, new-lines, etc until the end of the string or until a conversion specification is met.

A conversion specification comprises the ‘%’ symbol followed by symbols which control the formatting in the following way:

%[-][[0]width][.max]<char>

The arguments p1, p2, p3 and p4 are converted and formatted according to this conversion specification and appear in their respective positions.

The ‘-’ field forces the argument to be converted left justified within its field, rather than right justified.

The ‘[0]width’ field specifies the width of the field for the conversion (’width’ represents a decimal number). If the ‘0’ is given the field is padded with leading zeroes, otherwise blanks are used.

The ‘.max’ number specifies the maximum number of characters to be printed out on a string (%s) conversion.

The <char> specifies how the argument is to be converted for output:

%b	convert to binary number (1’s and 0’s)
%c	print out as ASCII char
%d	signed decimal number
%e	converts a negative integer input to a system error message
%o	octal number
%s	string
%u	unsigned decimal
%x	hexadecimal number

For all the conversions except ‘%s’ and ‘%e’ the arguments are printed out as numbers. The ‘%s’ conversion specification inserts the null-terminated string pointed to by the corresponding argument into the output stream. The ‘%e’ conversion specification takes a negative integer from the system error messages, and outputs a human readable string in its place. This makes error message handling very easy.

Examples:

system call:	printf("message");
d1	pointer to control string
output	message

Here d1 points to the null-terminated string "message", which will be printed out literally because it contains no conversion specifications.

system call	printf("value is %d", 10)	
	d1	pointer to control string
	d2	10
output	value is 10	

Here d1 points to the control string, d2 contains 10 (binary). The ‘printf’ function prints out characters from the control string until it encounters the conversion specification, where upon the first parameter is converted into a decimal string and printed out.

system call	printf("p1 is %d, p2 is %d, p3 is %d, p4 is %d",1,10,-5,12)
d1	pointer to control string
d2	1
a0	10
a1	-5
a2	12
output	p1 is 1, p2 is 10, p3 is -5, p4 is 12

The assembly code implementation of this is:

```
          move.l     #contstr,d1
          move.l     #1,d2
          move.l     #10,a0
          move.l     #-5,a1
          move.l     #12,a2
          move.l     #48,d0      * ‘printf’ sycall number
          trap       #7          * do the trap
          rts
contstr   dc.b       "p1 is %d, p2 is %d,
                     p3 is %d, p4 is %d",0
          end
```

Data formatting - sprintf

sprintf(buf, contstr, p1, p2, p3, p4)

d0	49	
d1	buf	pointer to output area
d2	contstr	pointer to null-terminated control string
a0	p1	first output argument
a1	p2	second output argument
a2	p3	third output argument
Return	nil	

This call formats the specified control string and data (as with ‘printf’) and leaves the output as a null-terminated string starting at the address buf, rather than printing out the characters.

Formatted output to a stream - fprintf

fprintf(handle, contstr, p1, p2, p3)

d0	120	
d1	handle	character device or file handle
d2	contstr	Formatting control string
a0	p1	First print argument
a1	p2	Second print argument
a2	p3	Third print argument
Return	nil	

Formats output according to contstr, p1, p2 and p3; simultaneously directing it to the output stream identified by handle.

String output to a character stream - fputs

fputs(handle, buf)

d0	121	
d1	handle	character device or file handle
d2	buf	null-terminated string to output
Return	0 (or negative error code)	

Puts the string at buf out through the selected stream. May return an error code if the handle refers to a file descriptor.

Line input from a character stream - fgets

fgets(handle, buf)

d0	123	
d1	handle	character device or file handle
d2	buf	buffer area for line
Return	buf	

Gets a null-terminated line from the input stream. The maximum length of the line is 512 characters. Strips off new-lines and line feeds, returning a line at a time. Returns a null string (first character in the buffer is zero) on end of file or any other file I/O error. buf must point to a buffer of at least 513 bytes.

If handle refers to a character device, an end-of-file condition exists when an end-of-file character is received. The end-of-file character is set by the 1616/OS command OPTION. It is normally set to $100 (no EOF character), but can be toggled to Ctrl D (04) by the Alt Del hotkey.

Character device miscellaneous - cdmisc

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

d0 133

Refer to the *Technical Reference Manual* for details of this very extensive system call for manipulating character devices. You would normally use the `chdev` program from the *Users Disk* rather than attempt to manipulate this call directly.

6
Video Output Calls

1616/OS's video drivers support both 320 and 640 column modes with a low-level 'windowing' facility which permits text and graphics writing within a portion of the screen whilst protecting the regions outside the window.

A terminal emulator is included, with various text attributes such as underline, bold, italic, subscript, superscript, highlights included. Full cursor positioning is available, with line insertion and deletion, backward scrolling, and a number of similar constructs. This eases the task of writing text manipulation routines, since all of these facilities can be accessed by using escape code, as described near the end of this chapter.

Introduction

Incarnations of 1616/OS before V3.0 used hardware scrolling of the video display. This was done by offsetting addresses within the 1616's video controller chip. Hardware scrolling gave good scrolling performance but made the direct addressing of video memory quite complicated. The clamour of complaints from the confused has resulted in hardware scrolling being dropped. Each pixel on a video page now occurs at a fixed offset relative to the start of the page, but scrolling was slower. This slow scrolling was fixed in Version 4.2, and users can gain direct access to the high speed low level video drivers, if required.

If no attributes are set in text for display, the video driver is much faster (up to ten times) in Version 4.2a. Best performance is for white characters on a black background, while bold, italic, etc., all slow down the display.

The 1616's video circuitry may display any of the sixteen 32k pages in the on-board memory. The pages start on 32k address boundaries. The normal display page is page 15 (start address normally = $78000). The 1616/OS video software can draw text and graphics on any of the display pages, whilst displaying any other (or the same) page.

If using Conal Walsh's EGA drivers (or indeed any higher resolution display) you will need to allocate additional video memory in your `mrdrivers` file. Users of additional memory boards should note that allocating all 512k of on-board memory to video will prevent certain programs (such as `vcon`) from operating correctly.

For many of the following video-specific character output functions it is better to print characters out using the '***putc***(0, ch)' system call, rather than '***putchar***'. This ensures that the characters are actually sent to the video character driver, thus overriding any I/O redirections.

Extensions to the graphics support are highly likely in the future (Andrew was looking at various window systems).

640 column mode:

In 640 column mode, each word (16 bit quantity) in the 1616's video display memory is represented as pixels on the screen in the following manner

Bits in one video word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
|	|													|	|

Index into
register file of
leftmost pixel

Index into
register file of
rightmost pixel

Each of the two-bit pairs in the top diagram are used as an index into the 1616's colour palette, the 4x4 bit register file, U4. The four x four bit words in the register file are written by the processor using the ***set_pal*** system call and allow the programmer to map each of the four colour combinations into one of the sixteen available colours.

For example: If value of bits 15 and 14 equals 0, and palette entry 0 is set to 5, then the colour of the displayed pixel is 5.

320 column mode

In this mode the four bit RGB and I colour signals are read directly from memory, rather than being read from the register file. The second diagram represents this.

In 320 mode the mapping is:

Bits in one video word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
|	|	|	|									|	|	|	|

Colour of
leftmost pixel

Colour of
rightmost pixel

The border colour may be any of 16 colours, independent of the video mode. If you use a multisync monitor, border colours may not display correctly, as some multisyncs use the border colour level to determine the black level for the entire display. This is not a fault of the Applix 1616. It occurs because most multisync displays are designed for IBM PC clones which can not alter their border colour.

Set/clear 640 column mode - set_640

set_640(mode)

d0 31

d1	mode	0 320 mode, 1 640 mode, 2: determine current mode
Return	current mode	

Sets the video resolution to 320 or 640 columns. If called with mode = 2, the current mode is returned (0 = 320 mode, 1 = 640 mode).

Many of the video and graphics system calls' internal functions work differently in the two modes, however the differences are almost totally transparent from an application software point of view.

For most applications you will have to change the video window size when going between these two modes, to reflect the different number of columns available (a call to ***def_wind*** with an argument of 0 once the mode has been set will do this).

Conal Walsh has provided an extensive range of EGA style video displays. An MRD is available in shareware to allow access to these additional modes, using modes 8, 10 and 12.

Set the video display page - set_vdp

set_vdp(page)

d0	32	
d1	page	new display page
Return	nil	

This call selects one of the 1616's sixteen 32 kbyte video display pages for the source of video display output. Bounce through these to find out what code looks like when it is in memory! Note that you can set the display page to larger than 32 kbytes, by using an MRD. This is used for EGA and denser displays.

Set the video software access page - set_vap

set_vap(page)

d0	33	
d1	page	new access page
Return	nil	

This call selects the 32k video page which is henceforth to be used for software accesses. All video output calls alter the page which is selected here: the displaying of the page is quite independent of the software access.

Set the video text foreground colour mask - set_fgcol

set_fgcol (colmask)

d0:	34	
d1	colmask	colour mask
Return	nil	

Character display data is created by taking a 16 bit (one row of a character, 8 pixels) word from the character set lookup table, called ‘charpattern’ here and calculating the following:

data = (charpattern and fgmask) or ((not charpattern) and bgmask)

The resulting pattern is written into the display RAM.

The character set patterns are 16 bits wide, with an 8 x 8 font implemented by setting neighbouring bits in the 16 bit word to the same value. Pixels which are ‘on’ in the character shapes are represented by a ‘11’; pixels which are off are represented by a ‘00’.

In 640 mode the foreground and background colour masks may be considered to be a group of eight bit-pairs, each pair corresponding to a pixel colour (bits 14 15 correspond to the leftmost pixel of a character).

In 320 mode the masks are a group of four x four-bit groups. Each group determines the character colours in 320 mode.

The effect of all this is that the foreground colour mask determines the colour of the character foreground and the background mask determines the colour of the background!

Useful values for the masks in 640 column mode are:

colour = 0	$0000
colour = 1	$5555
colour = 2	$aaaa
colour = 3	$ffff

In 320 mode:

colour = 0	$0000
colour = 1	$1111
colour = 2	$2222

etc.

Set the video text background colour mask - set_bgcol

set_bgcol(colmask)

d0	35	
d1	colmask	16 bit background colour mask

Return	nil	

The background of a character is the part which is normally black; varying it provides different text appearances: see ***set_fgcol***, above.

Set the video border colour - set_bdcol

set_bdcol(col)

d0	36	
d1	col	colour
Return	nil	

Sets the video border colour to col. The borders of the screen are those parts of the phosphor which are outside the normal viewing area.

The colour may be in the range 0 - 15.

Beware of some multisync monitors setting their black level to the border colou. This can be a real pain for colourful people.

Set a palette entry - set_pal

set_pal(palpos, col)

d0	37	
d1	palpos	palette index
d2	col	colour at that index
Return	nil	

This system call sets up one of the 1616's video palette entries. See the start of this section for a description of how video data indexes into the register file.

The palette index, palpos is in the range 0 - 3.

The colour, col is in the range 0 - 15.

Get a pointer to a character shape definition - rdch_shape

rdch_shape(charno)

d0	38	
d1	charno	ASCII code of character
Return	pointer to character shape table	

Returns a pointer to 1616/OS's internal representation of the character whose ASCII code is in charno. The pointer points to 8 x 16 bit words. For example, here is the shape for the character 'A':

bit	15					to										0	
word 0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	Row zero
word 1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	Row one
word 2	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	Row two
word 3	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	Row three
word 4	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	Row four
word 5	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	Row five
word 6	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	Row six
word 7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Row seven
pixel	7		6		5		4		3		2		1		0		

The neighbouring bit-pairs for each pixel are always set, so that the foreground and background masking works correctly.

This system call is provided so that you may alter and install character shapes. All characters from $00 to $ff are valid.

Redefine a character shape - def_chshape

def_chshape(charno, defptr)

d0	39	
d1	charno	ASCII code of character to redefine
d2	defptr	pointer to new definition
Return	nil	

Moves the 16 byte character shape definition pointed to by defptr so that it overwrites the old definition. Your new shape, charno, is used henceforth.

If the first argument (in d1) is -1, a pointer to the raw EPROM character set, in uncompressed form, is returned (from Version 4.0b).

Select a new video character set - newchset

newchset(ptr)

d0	130	
d1	ptr	pointer to new character set
Return	nil	

ptr points to a replacement character set which may be up to 4 kbytes in size. The character set consists of up to 256 bitmaps such as that above, arranged in ASCII order.

If ptr is zero, the system restores the internal, default character set (very handy when you really stuff it up).

Define a video window - def_wind

def_wind(wind)

d0	40	
d1	windptr	pointer to window definition structure
Return	usually garbage (formerly pointer to current window definition)	

This system call defines a rectangular window for subsequent text and graphics use. The window is effectively a smaller screen. All scrolling, cursor positioning, graphics commands, etc. are relative to the window, rather than to the 1616's entire video display. In fact the normal 80 x 25 display is internally treated as just another window, 80 characters by 25.

If this call is made with windptr equal to zero then the system installs a default window for the current mode (25 * 40 for 320 mode, 25 * 80 for 640 mode).

If this call is made with windptr equal to 1 then a pointer to the window structure which the system is currently using is returned. No alterations are made.

A window is defined to the system by passing a pointer to a data structure comprising of 8 16-bit words:

```
x_start   dc.w    1       * Start coordinates, relative to top left of
y_start   dc.w    1       * 80x25 screen
xend      dc.w    1       * End co-ordinates + 1, relative to 80x25
yend      dc.w    1       * screen
bg_col    dc.w    1       * Window background colour mask
fg_col    dc.w    1       * Window foreground colour mask
curs_x    dc.w    1       * Current cursor relative x position
curs_y    dc.w    1       * Current cursor relative y position
```

These words are used as follows:

xstart, ystart

These define the position top left character of the window in the entire video display. The very top left character has coordinates of (0, 0).

xend, yend

These define the position of the bottom right-hand corner of the window in the entire display. The character at absolute position (xend, yend) is in fact outside the window. The character at (xend-1, yend-1) is just inside the window. The total size of the defined window is given by (xend-xstart) x (yend-ystart).

fg_col, bg_col

The foreground and background colours used for text inside this window.

curs_x, curs_y

The relative position of the cursor in this window. The cursor is both the place where the blinking square appears and the place where the next printable character is put. This cursor position is relative to the window. The character at (curs_x, curs_y) appears at the absolute screen position (xstart+curs_x, ystart+curs_y).

This 8 word data structure is located within your program's data areas. The cursor position and colour masks are updated by the system as they change, so you may select a different window structure at a different screen position and later come back to the original one, with the necessary information preserved.

When a window is selected it becomes the 'current window'. All cursor positioning commands occur relative to the window's top left corner; the colour commands affect this window alone.

Multiple windows are implemented by having several such data structures in memory, each with different contents (particularly different positions!). The windows are selected when needed and characters are printed out and graphics functions may be performed.

When selecting a window for the first time with a pointer to a data area which has not been initialised, first set xstart, ystart, xend and yend (these should not be varied for the life of the window). Initialise the background colour to $0000 (optional) and the foreground colour to $ffff (optional) and set curs_x and curs_y to 0.

As an alternative to clearing curs_x and curs_y, you can print out a control-L (clear screen command) immediately after the window has been defined. This will fill the window with your background colour (here the system is clearing the screen where the current screen is a window). It will also move the cursor to 0, 0.

Move video window contents - move_wind

move_wind(buf, mode)

d0	42	
d1	buf	pointer to memory buffer
d2	mode	move mode
Return	nil	

This system call is used for moving the data contained in the current window to and from main memory.

mode = 0

The data in the window is swapped with the data pointed to by buf.

mode = 1

The data in the window is moved to memory pointed to by buf.

mode = 2

The data pointed to by buf is moved to the window.

The data in the window is treated in its raw, bit-mapped mode. You may freely move data in and out of windows, permitting overlaying. The only restriction is that once the data from within a window has been pulled into main memory it must be put back into a window with the same horizontal and vertical sizes. It may be written back to a different position.

The amount of free memory required at buf is determined by the size of the current window:

buffer size (in bytes) =

(xend - xstart) * (yend - ystart) * 16 (in 640 mode)

(xend - xstart) * (yend - ystart) * 32 (in 320 mode)

This function is quick!

Fill the video window - fill_wind

fill_wind(col)

d0	44	
d1	col	colour to fill with
Return	nil	

Entirely fills the current window with the lower 16 bits of col.

Get physical video addresses - vid_address

vid_address(x, y)

d0	41	
d1	x	x coordinate
d2	y	y coordinate
Return	byte address	

This system call is provided to facilitate direct manipulation of the video display RAM by application programs.

The arguments x and y are the coordinates of a pixel on the screen, relative to the top left corner of the screen, not the current window. Hence x will be in the range 0 - 639 and y will be in the range 0 - 199. This range holds true for both 320 and 640 mode: in 320 mode you will have to multiply the desired pixel's x coordinate by 2 before making this call.

This call returns a pointer to the actual byte which the 1616's video circuitry reads from memory to produce the pixel. In 640 mode there will be four pixels contained within this byte (2 in 320 mode); it is up to you to resolve the actual bit position within the byte.

The translation from screen coordinates to physical addresses is quite complicated in the 1616, but less so than under pre-3.0 versions of the operating system. If you are doing intensive direct screen I/O the following steps are suggested:

- Set up the desired video mode (320 or 640)
- Call ***def_wind*** with an argument of 0 to use the whole screen
- Call ***fill_wind*** with the desired background colour.
- Make 200 calls to ***vid_address*** with x = 0 and y incrementing, setting up a local table of pointers to the start of each line in your data areas.
- Do all your video access by offsetting from these pointers

Low level video character drawing - rawvid

rawvid(row, col, ch, fgmask, bgmask)

d0	43	
d1	row	character row number
d2	col	character column number
a0	ch	ASCII code
a1	fgmask	foreground colour mask
a2	bgmask	background colour mask
Return	nil	

This call writes the character ch into the video display at row row, column col. The current window setting is ignored. This is a low-level entry point which has the advantage of being re-entrant: if you wish to print out characters from within an interrupt subroutine this is the way to do it. In Version 4, the top byte of the foreground colour mask (passed in a1) appears to set the text attribute (bold, underline, etc) for characters printed (I don’t believe Andrew has admitted to arranging that, so it may not always be safe).

Alter cursor mode - scurs_mode

scurs_mode(rate, enable, mask)

d0	45	
d1	rate	cursor flash rate
d2	enable	cursor enable
a0	mask	cursor bit mask
Return	nil	

This system call permits manipulation of the video cursor. The 1616 video cursor is implemented under software, using the system vertical sync interrupt timer.

The cursor has three attributes: the rate at which it flashes, whether or not it appears and its shape. These may all be altered with this system call.

rate

This argument sets the cursor flash rate. The resulting rate is (25/rate) flashes per second. If you don't want to alter the flash rate, pass a zero for rate.

enable

If enable = 0, retain current cursor mode.
If enable = 1, turn cursor off.
If enable = 2, re-enable flashing cursor.
If enable = 3, set rate, enable and mask to default state.
If enable = 4, return current rate setting.
If enable = 5, return current enable setting.
If enable = 6, return pointer to current cursor mask.

There is also a block cursor mode, operated by enable values of 7, 8, 9. Some may require rate and/or mask to be 0.
If enable = 7, set cursor to block mode.
If enable = 8, set cursor to flashing block.
If enable = 9, return non-zero if block mode already selected.

mask

pointer to eight 16-bit words which are exclusive-OR'ed with the character under the cursor at interrupt time. The data at mask is copied into internal 1616/OS data areas. If mask = 0 (nil pointer) then the blink mask remains unaltered.

The initial, default conditions (those which are set up by enable = 3) are:

rate = 12
enable = 2
mask = 8 * $ffff (block cursor)

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 include to facilitate the writing of a mouse driver. It is detailed (except for the embarrassing bits) in the *Technical Reference Manual*.

Video driver escape codes

The video display may be manipulated from within an application program by sending ‘control codes’ and ‘escape codes’ to the video driver. Control codes are simply some of the ASCII characters less that 32. Escape codes are done by printing an ‘ESC’ character (ASCII code $1b hex, 27 decimal), followed by an appropriate character sequence. Remember that all these codes are case sensitive.

Many new escape codes have been added to the terminal driver. It is now very similar to the Televideo 950 terminal. The major difference is that the 1616 has 25 displayed lines, rather than 24. The driver supports underlining, italics, subscript, superscript and bold characters, all derived from the internal character set, on the fly.

`Option 18` sets the video output mode.

If bit 0 is set, all escape code sequences are ignored: the characters are simply printed out.

If bit 1 is set, all control characters remove their special meaning, even carriage return, linefeed, etc. Good for inspecting files from foreign machines.

Setting bits 0 and 1 puts the terminal in a sort of monitor mode, enabling you to inspect exactly what characters are coming out.

If bit 2 of `option 18` is set, the terminal emulation follows the TVI950 terminal emulation more closely. Specifically, if this bit is clear, then control-L clears the screen, and tabs overwrite characters. If this bit is set, then control-L moves the cursor forward, and tabs are non-destructive: they simply move the cursor out to the next tab stop, without writing anything on the screen.

The driver suppresses linefeeds after word wraps. This means that if a character is placed in the rightmost column, and the next character is a linefeed (or a carriage-return linefeed), then the linefeed is ignored. This is because the placing of a character in the last column will have caused a linefeed anyway. One consequence of this behaviour is that files whose lines are exactly eighty characters wide can be displayed without getting blank lines inserted.

All the escape codes operate relative to the current window.

Control codes

^@	The null character is ignored. ($0)
^I	Tabs move the cursor to the next tab column which is an even multiple of eight. If bit 2 of `option 18` is clear, spaces are drawn. If it is not clear, no spaces are drawn. ($9)
^B	Beep the speaker, different tone. ($2)
^G	Beeps the speaker. ($7)
^J	Linefeed. ($A or .10)

^K Moves the cursor up, if it is not at the top row. ($B or .11)

^L If bit 2 of option 18 is clear, it clears the screen. Otherwise move the cursor forward, if it is not in the last column. ($C or .12)

^M Carriage return. ($D or .13)

^V Moves the cursor down, if it is not at the bottom row. ($16 or .22)

^Z Clear the screen and home the cursor. ($1A .26)

^^ Ascii code $1e or .30 : homes the cursor.

Escape sequences

ESC = (row+32) (col+32) Positions the cursor. ($1B, $3D, row + $20, col + $20).

ESC) Start highlighting. ($1B, $29) Set current foreground colour to 2 in 640 mode, 10 in 320 mode. Current background becomes 1 in 640 mode, 5 in 320 mode.

ESC (End highlighting. ($1B, $28) Set current foreground to 3 in 640 mode, 15 in 320 mode, background to 0.

ESC * Clear the screen, or current window, home the cursor. ($1B, $2A).

ESC B (value+32) Sets the background colour to ‘value’. ($1B, $42, col + $20) Only values allowed are 0 to 3 in 640 mode, 0 to 15 in 320 mode.

ESC b Visible bell. ($1B, $62) This code generates a beep which has a lower frequency than that produced by a ^G.

ESC E Insert a line at the current one: all lines below the current line are moved down one, clears current line. ($1B, $45)

ESC F (value+32) Sets the foreground colour to ‘value’. ($1B, $46, col + $20) Only colours 0 to 3 in 640 mode, 0 to 15 in 320 mode.

ESC G 1 Sets subscript mode.

ESC G 2 Sets superscript mode.

ESC G 4 Sets bold mode.

ESC G 8 Sets underline mode.

ESC G @ Sets italic mode.

ESC G 0 Clears subscript, superscript, underline, bold and italic modes.

ESC I Back tab: moves the cursor back to the previous tab stop, provided it is not in column 0.

ESC j Reverse scroll display.

ESC M (from+32) (to+32) Copies the contents of a line from one row to another. ($1B, $4D, from + $20, to + $20).

ESC P (position+32) (value+32) Writes 'value' into the 1616 video pallette at 'position'. ($1B, $50, position + $20, value + $20).

ESC Q Character insert. Moves every character to the right of the cursor one character to the right.

ESC q Enter insert mode. In this mode every printed character causes all the characters to its right to be moved right. The rightmost character is lost.

ESC r End insert mode. ($1B, $72).

ESC R Delete the current line. ($1B, $52) Moves all the lines below the current one up one line. Clears bottom line to current background colour.

ESC S (value+32) Sets the border colour to 'value'. ($1B, $53, value + $20).

ESC t Clears from cursor to the end of the line. ($1B, $74).

ESC T Clears from cursor to the end of the line. ($1B, $54)

ESC W Delete character. All characters to the right of the cursor are moved left one position, deleting the character under the cursor.

ESC Y Clears from cursor to the end of the screen.

ESC **.** 0 Cursor off.

ESC **.** 1 Flashing block cursor.

ESC **.** 2 Steady block cursor.

ESC **.** 3 Flashing underline cursor.

ESC **.** 4 Steady underline cursor.

ESC 0x03 *nn* Delay of *nn* ticks

7
Graphics System Calls

Introduction

The graphics system calls are relatively straightforward: pixels, lines, etc may be drawn relative to a window (the current window) or relative to the screen as a whole. Graphics functions which operate relative to the entire screen are said to draw on the 'absolute screen'.

If graphics operations are being performed relative to the current window then the pixel at (0,0) corresponds to the top left pixel of the top left character in the current window. For graphics purposes the current window may be considered to extend from (xstart*8, ystart*8) to (xend*8 - 1, yend*8 - 1). These are pixel coordinates, not character coordinates - characters take eight pixels in both directions.

When using graphics functions, don't forget to set your foreground and background colours prior to making the line drawing or fill calls. Nothing is less visible than a black line drawn on a black background!

Area fill routine - fill

fill(x, y, val)

d0	47	
d1	x	x coordinate of start point
d2	y	y coordinate of start point
a0	val	value to fill with or pointer to texture table
Return	0 or -1 if fill failed	

This is an area fill routine based upon Jeremy Fitzhardinge's algorithm.

fill sets all the pixels in an area of any shape. The value of the pixel at (x, y) is called the background colour. Its value is read and the fill routine visits and changes every pixel of the same value in the currently defined window to which a path can be found.

x and y are a pixel position relative to the current window. If val is in the range 0-15 it is assumed to be the value with which to fill the area. If val is outside this range it is assumed to point to a table of sixteen 32 bit long words which represent a bit-mapped texture pattern with which to perform the fill.

If the fill is non-textured (val in the range 0-15) ***fill*** checks that val is not the same as the background colour, which would result in no filling at all. A value of -1 is returned in this case.

In 640 pixel mode the texture table represents an array of 16x16 pixels, 2 bits per pixel. In 320 pixel mode it represents a table of 8 horizontal by 16 vertical pixels, 4 bits per pixel.

Before performing a textured fill, this system call checks that none of the pixels represented in the texture table have the same value as the background colour, which would cause ***fill*** to never terminate. A value of -1 is returned if this happens.

The fill routine uses 16000 bytes of stack space in which to store its backtracking information. A very complicated fill may cause this stack to overflow. If this occurs the fill terminates and a negative error code is returned.

Raw graphics point draw - rset_pel

rset_pel(x, y, val)

d0	50	
d1	x	x coordinate
d2	y	y coordinate
a0	val	value to plot
Return	nil	

Plots a point directly on the screen, relative to extreme top-left pixel, ignoring the current window.

In 320 mode the four least significant bits of val are written to the selected pixel, hence val is the desired pixel colour.

In 640 mode the two least significant bits of val are written to the selected pixel, hence val is the desired index into the colour pallette register file.

Windowed graphics point draw - set_pel

set_pel(x, y, val)

d0	51	
d1	x	x coordinate
d2	y	y coordinate
a0	val	value to plot
Return	nil	

Plots the pixel as with ***rset_pel*** above, except that x and y are relative to the current window, not to the entire screen. If the point lies outside the current window it is not plotted. Note that d2 (y pointer) may be incremented during this call, so it has to be reset to the correct value prior to the next call.

Raw graphics line draw - rline

rline(x0, y0, x1, y1)

d0	52	
d1	x0	x coordinate of line start
d2	y0	y coordinate of line start
a0	x1	x coordinate of line end
a1	y1	y coordinate of line end
Return	nil	

This system call is no longer supported. Use ***drawline***.

Windowed graphics line draw - drawline

drawline(x0, y0, x1, y1)

d0	53	
d1	x0	x coordinate of line start
d2	y0	y coordinate of line start
a0	x1	x coordinate of line end
a1	y1	y coordinate of line end
Return	nil	

Draws a line within and relative to the current window. Any pixels which lie outside the current window are not plotted. The line is drawn in the current graphics colour.

Raw pixel read - rread_pel

rread_pel(x, y)

d0	54	
d1	x	x coordinate
d2	y	y coordinate
Return	pixel value	

Reads the value of a pixel positioned relative to the absolute screen.

In 320 mode a number between 0 and 15 (the colour) is returned. In 640 mode a number between 0 and 3 (the colour pallette register file index) is returned.

Windowed pixel read - read_pel

read_pel(x, y)

d0	55	
d1	x	x coordinate
d2	y	y coordinate
Return	pixel value	

Reads the value of a pixel positioned relative to the current window.

Set graphics colour - sgcol

sgcol(colour)

d0	56	
d1	colour	line draw colour
Return	nil	

Lines drawn after this system call are drawn with the specified colour. The colour may be in the range 0-3 (640 mode) or 0-15 (320 mode).

Set graphics background colour - sgbgcol

sgbgcol(colour)

d0	57

Set graphics line texture mask - sgtexture

sgtexture(val)

d0	58

These two system calls are no longer supported.

Raw circle draw - rcircle

rcircle(x, y, radius)

d0	59	
d1	x	x coordinate of centre
d2	y	y coordinate of centre
a0	radius	circle radius

Return	nil

Draws a circle positioned at x, y on the absolute screen with radius radius. The circle is drawn in the current graphics colour.

Windowed circle draw - circle

circle(x, y, radius)

d0	60	
d1	x	x coordinate of centre
d2	y	y coordinate of centre
a0	radius	circle radius
Return	nil	

Draws a circle positioned at x, y relative to the current window with radius radius. The circle is drawn in the current graphics colour. Any points which lie outside the current window are not plotted.

Set graphics dot draw mode - sdotmode

sdotmode

d0	61	
d1	mode	0 = write 1 = OR 2 = AND 3 = XOR 4 = read
Return	Current mode	

This system call assigns the manner in which all the graphics output system calls draw pixels. Essentially there are four internal pixel setting modes (0-3), and one mode to read back the current setting (4).

mode 0 write - pm_write
This is the default mode. Every time a pixel is to be drawn, the new value overwrites the old one in video RAM.

mode 1 OR - pm_or
The new pixel data is logically OR'ed with the data presently in the video RAM. Thus, and '1' bits in the new pixel set individual bits within a pixel, rather than an entire pixel.

mode 2 AND - pm_and
The new pixel data is complemented and logically AND'ed with the data presently in video RAM. Any '1' bits in the new pixel data clear the corresponding bits in the pixels in video RAM.

mode 3 XOR - pm_xor
The new pixel data is logically OR'ed with the data currently in video RAM, so that a '1' bit in the new pixel data causes the corresponding bit in the video RAM to be complemented.

mode 4 read - pm_read
Simply reads back the current setting of the dot mode.

The dot mode may be used for addressing individual bit planes in the video RAM. In 320 mode you can individually address four bit planes, by setting the foreground colour to 1, 2, 4 or 8. You draw lines in the bit plane by calling ***drawline***, ***set_pel***, etc., with OR mode set. The lines may be erased again (leaving any data in other bit planes undisturbed) by setting AND mode, and redrawing them with the same start and end points, and the same foreground colour.

8 Hardware Control Calls

Introduction

These are a group of system calls which may be used to perform various manipulations upon the 1616's hardware without having to resort to directly accessing I/O ports.

These calls allow easy selection and use of the analog to digital and digital to analog facilities, including tone generation for playing sounds, and testing for completion of a previous tone. There are calls for testing for DIP switch and joystick button settings, and also for programming the serial ports and the video controller chip. For more visual effects, there is an LED you can blink (or even pulse width modulate, if you really want to go to the trouble).

A later version of the 1616/OS will include calls for easy access to A-bus peripheral boards.

Select an analogue input - anipsel

anipsel(ipnum)

d0	70	
d1	ipnum	analogue input channel number
Return	nil	

Uses the passed number to switch one of the analogue inputs into the analogue-to-digital converter comparator input.

Input channel 7 selects the joystick X direction potentiometer. Input channel 6 selects the joystick Y direction potentiometer. Input channels 5 to 0 select general purpose analogue inputs AI5 to AI0 on pins 32 to 27 on the User I/O connector.

Select an analogue output - anopsel

anopsel(opnum)

d0	71	
d1	opnum	analogue output channel number
Return	nil	

Selects one of the four analogue outputs (driven by integrated circuit U23, a dual 4 channel analogue multiplexor, 4052). When a particular analogue output is selected, all the others hold their previous voltages (for a limited period) on their holding capacitors. Analogue outputs 2 and 3 (shown in schematic as 0 and 1) are available on pins 33 and 34 of the User I/O connector, after buffering by an LM324 operational amplifier. The other outputs go to the right and left loudspeaker channels, via a two or four watt stereo amplifier.

output usage channel

0	Right sound channel
1	Left sound channel
2	Analogue output 0
3	Analogue output 1

Disable analogue outputs - anopdis

anopdis( )

d0	72
Return	nil

Disables all four analogue outputs, so that all channels hold their current voltages (for a while). You should disable outputs prior to doing an A to D conversion using ***adc***.

Perform analogue to digital conversion - adc

adc( )

d0	73
Return	converted value

This call performs a successive approximations analogue-to-digital conversion of the current analogue input source (selected with the ***anipsel*** call).

The returned value will be in the range 0 - 255. A value of 0 indicates an input voltage of approximately -2.2 volts; a value of 255 indicates a voltage of approximately +2.2 volts.

Note that the A/D conversion routine changes the output of the DAC, so it may be desirable to disable all the analogue outputs using ***anopdis*** before performing A/D conversions.

Perform digital to analogue conversion - dac

dac(val)

d0	74	
d1	val	DAC output value
Return	nil	

Writes the passed value out to the DAC. If one of the analogue output channels is selected, then its level will change to reflect the new value.

Set/clear LED - set_led

set_led(val)

d0	75	
d1	val	0 LED off, 1 LED on
Return	nil	

Turns the 1616's status LED on or off. Write code to pulse width modulate it for special effects.

Play a waveform - freetone

freetone(table, tablen, length, preload)

d0	76	
d1	table	pointer to sound table
d2	tablen	length of table (in bytes)
a0	length	number of bytes to output
a1	preload	VIA timer1 preload constant
Return	nil	

This system call takes a table of bytes pointed to by table and sequentially writes them out to the DAC. To produce a sound waveform, select one of the speaker analogue output channels (using ***anopsel***) before performing this call. This system call may be used for general purpose waveform output from any of the analogue output channels. If the length field is $fffffff the duration of the waveform will be very long indeed - effectively infinite for most applications.

The tablen argument refers to the length of your table (in bytes).

The length argument is the number of bytes which are to be sent from your table to the DAC. If length is greater than tablen then the software recirculates through the table (it 'wraps around').

The preload argument is used to set the period between samples. It is passed to the ***ent1ints*** system call to generate an interrupt stream.

Since the number of samples which are sent is length we can calculate the actual duration of the free tone from:
tone duration = length * ((2 * preload) + 3.5) / 750,000 seconds.
For an 11 kHz sound file, I calculate that 32 should be about right.

This system call returns immediately with the interrupt stream and some other values initialised. Your program may continue executing (more slowly!) while the interrupts continue. Since the DAC output data is being obtained from the table in your data space you should not alter it until the ***freetone*** has ended (unless you specifically want to). When the tone has completed the interrupts are disabled. The time during which the waveform is being played may be used for building the next table.

The ***fttime*** system call is provided for determining the current state of the ***freetone*** interrupt code and may be used to poll for the completion of the previous tone.

The waveform output may be prematurely halted by using the ***dist1ints*** system call. It may be restarted or changed by performing another ***freetone*** before the first has ended.

Note that the DAC produces its most negative voltage with an input of $00 and its most positive voltage with an input of $ff, so sound waveforms should have a mean of $80 to get best results and the widest possible dynamic range.

Return time left for freetone completion - fttime

fttime( )

d0	77
Return	counts to completion

Returns the number of VIA timer1 interrupts left until completion of the last free tone output. When this call returns 0, the ***freetone*** has ended. Please check before starting anther ***freetone***, or the results won't be what you expect. If playing soundfiles from the command line, look up the `wait` command!

Read input port - rdiport

rdiport( )

d0	78
Return	input port value

Returns the byte read from the 1616's input port (integrated circuit U19). This port is the joystick buttons, plus the four DIP switches. See below for a more useful (albeit more complicated) version of this call.

Read time-accumulated input port - rdbiport

rdbiport( )

d0	79
Return	accumulated value

There is a vertical sync interrupt ISR within 1616/OS which reads the input port every 20 milliseconds and accumulates a logical OR and a logical AND of the readings. This system call returns the OR and AND accumulators and reinitialises them.

This is specifically designed for catching quick presses of the joystick buttons and remembering the press for programs which cannot afford to waste time continually polling the buttons. If, for example, a joystick button input went from high to low and then high again, the AND accumulator would retain a zero in the relevant bit position.

Bits 0-7 of the returned long word represent the OR accumulator; bits 8-15 represent the AND accumulator.

In general, to use this function you must initially call it once to initialise the accumulators, then discard the result. From this point onwards, if a call to ***rdbiport*** returns with bit 2 of d0 set, then a high has been detected on PB0 (push button zero) - it may still be high and will have to be checked for a release. If the call returns with bit 10 of d0 clear then a low has been detected on PB0.

Reprogram a serial port - prog_sio

prog_sio(chan, spptr)

d0	82	
d1	chan	0 for channel A, 1 for channel B
d2	spptr or 0 or 1	Pointer to serial channel initialisation structure
Return	0 (-1 if bad parameters)	

Reprograms one of the 1616’s Zilog 8530 SCC serial channels. spptr must point to the following five word data structure:

baudrate	dc.w 1	* Actual baud rate (eg, #9600 decimal)
rxbits	dc.w 1	* 0: 5 bits, 1: 6 bits, 2: 7 bits, 3: 8 bits
txbits	dc.w 1	* 0: 5 bits, 1: 6 bits, 2: 7 bits, 3: 8 bits
parity	dc.w 1	* 0: None, 1: Odd, 2: Even
stopbits	dc.w 1	* 0: 1 stop, 1: 1.5 stop, 2: 2 stop

If any of the words are outside the indicated range an error code of -1 is returned. The serial receive routine adds masks ANDing with $7F, $3F or $1F, when programmed for 7, 6 or 5 bits. This was added in V3.2b to keep Andrew McNamara happy!

If the syscall is entered with d2 containing 0, then the call returns a pointer. The pointer is to a data structure identical to that used by spptr, which will contain the current settings of the specified serial channel. This makes it much easier to determine the current serial port settings.

If the syscall is entered with d2 containing 1, the call returns the address of the SCC structure for SCC channel chan.

The serial drivers were extensively reworked as at 1616/OS Version 4.2a, in association with extensive changes to character devices in general (see the ***cdmisc*** syscall). The driver now supports up to 4 SCC (8530) serial chips, for potential use with an expansion card containing an extra three SCC chips.

Note that detection of a break condition on the SCC is only possible when the receiver is enabled. i.e. DCD is asserted, or DCD is being ignored by having the SCC ignore hardware flow control.

There is a device associated with each SCC channel, as described in the header file `scc.h` on your 4.2 User Disk.

CRTC initialise - crtc_init

crtc_init(mode, ptr)

d0	140	
d1	mode	programming mode
d2	ptr	Pointer to 14 byte memory buffer
Return		None

mode = 0
Move the 14 bytes pointed to by ptr into 6545 CRTC registers 0 to 13.

mode = 1
Move the 1616/OS shadow register image of the 6545 CRTC registers to memory pointed at by ptr.

mode = 2
Restores the 6545 CRTC and its OS image to the default settings.

There is now a shadow register set for the CRTC, and the shadow registers can be manipulated via this syscall. Obviously you use mode 1 first, hen modify th values obtained, and then write them to the CRTC using mode 0 ... unless you like calculating CTRC settings

9
Short Form Call List Appendix A

This Appendix is now included in the *Quick Reference Manual*. Too hard to keep it up to date in two places. A fairly complete list of syscalls is included in the index.

10
Error Messages Appendix B

Most of the error messages which come out of 1616/OS are self explanatory. There is an amount of internal consistency checking and protection in version 3 and 4 of the OS, and violations of these can produce error messages which need more interpretation.

The other error messages fall into two categories: internal errors, and warnings, and are listed after theblock and memory errors.

Block and memory errors

Error code		Meaning
-1	ffff ffff	Unknown, general error
-2	ffff fffe	Disk is write protected
-3	ffff fffd	Invalid block driver number on block I/O system call
-4	ffff fffc	No space to install block device driver (8 max)
-5	ffff fffb	I/O error: bad disk, door open, RAM disk block checksum failure
-6	ffff fffa	Invalid block requested on block I/O call
-7	ffff fff9	Disk full
-8	ffff fff8	Invalid file handle on file I/O system call
-9	ffff fff7	Bad file name
-10	ffff fff6	File full (formerly 512k, obsolete)
-11	ffff fff5	File not currently open: possibly a bad file handle
-12	ffff fff4	File not open for reading
-13	ffff fff3	File not open for writing
-14	ffff fff2	File open for reading (not writing)
-15	ffff fff1	File open for writing (not reading)

-16	ffff fff0	Out of FCB’s - too many files open (16 max)
-17	ffff ffef	File not found
-18	ffff ffee	Attempted to read beyond end of file
-19	ffff ffed	Duplicate filename would result from ***rename***
-20	ffff ffec	Invalid argument: some nonsense passed to a file I/O call
-21	ffff ffeb	Attempted to seek beyond end of file
-22	ffff ffea	Seek not allowed - file is write-only (obsolete)
-23	ffff ffe9	File currently open
-24	ffff ffe8	Memory allocation failure: out of memory
-25	ffff ffe7	Directory: file-only operation performed upon a directory
-26	ffff ffe6	Not directory: directory-only operation on a file
-27	ffff ffe5	Tried to delete a non-empty directory
-28	ffff ffe4	Directory is full
-29	ffff ffe3	File is locked
-30	ffff ffe2	Bad magic number in relocatable file header
-31	ffff ffe1	User interrupt killed program
-32	ffff ffe0	No permission for file access
-33	ffff ffdf	Exec level too deep
-34	ffff ffde	Too many processes
-35	ffff ffdc	Exited due to kill
-36	ffff ffdb	Read/Write pipe with other end closed
-37	ffff ffda	Invalid PID passed to system call
-38	ffff ffd9	Code $%x: No error
-39	ffff ffd8	Error code -%d
-40	ffff ffd7	No error
-41	ffff ffd6	Unknown error

Internal errors

The system responds to an internal error by printing the following message:

```
Internal error N1: call PC = $N2. N3 N4
```

Where N1 is the internal error number, N2 is (perhaps) the program counter value at which the internal error occured. N3 and N4 are additional information, presented as hexadecimal numbers. After this message is displayed, the system halts, and needs to be restarted.

If an internal error can be reproducibly and inexplicably generated, please send APPLIX details of how to produce the error, so it can be investigated.

The implemented internal errors are listed below:

Internal error 1 and 2

Inconsistency in the memory manager (***getmem***), possibly due to memory corruption.

Internal error 3, 4, 5 and 6

Inconsistency in the memory manager (***freemem***), possibly due to memory corruption.

Internal error 7

Inconsistency in the memory manager during automatic freeing of a program's memory on return from ***exec***.

Internal error 10

A request was made to ***getmem*** for more memory than was available. This internal error may be disabled with OPTION 5. If OPTION 5 is disabled, ***getmem*** returns an error code to the calling program when out of memory, rather than generating an internal error.

Internal error 100, 101 and 102

Problems involving memory allocation and freeing in the line editor last-line recall management. Possibly caused by memory corruption.

Internal error 400

The file system code decided to deallocate a block which lies beyond the range of the disk. Probably caused by a corrupted file system. Use `fscheck.xrel` to repair it.

Internal error 500

In the ***callmrd*** system call a bad MRD header was encountered. This means that the memory reserved for MRDs has been corrupted. The system must be cold booted to recover.

Warning messages

These messages are produced when something unpleasant has been detected.

Suppressed write to block N1 on DEV [N2]

This message comes out of the ***blkwrite*** system call when an attempt has been made to write to the system block N1 on the device identified by DEV. The second number $N2 is the address from which the block was intended to be written. The system disables writing to the system blocks unless OPTION 8 has been set.

Panic: out of memory. PC = $N1

Yes, well. A call to ***getmem***, with OPTION 5 turned off, returned an error message during an attempt to allocate storage for the current directory cache. Solder in more RAM chips.

Released block N1: already free

During an attempt to deallocate block number N1 on a file system, it was discovered that the block was already free. This indicates an inconsistency in the file system. Run `fscheck.xrel` on the disk immediately to attempt to repair the disk.

Odd load address

The system was asked to load an `.exec` file to an odd address (68000 processors don't execute code at odd addresses).

Bad header magic

The magic bytes in the header of an `.xrel` file were not present. Caused by a misnamed or corrupted file.

Truncated xrel file

Something is wrong with the relocation table at the end of an `.xrel` file.

freemem(N1)

Somebody called ***freemem*** with an argument of N1 (N1 is in hexadecimal). The address N1 does not correspond with the memory manager's information, so either N1 was never returned by ***getmem*** or ***getfmem***, or some corruption of memory has occured.

freemem(0xN1)[OxN2] returns 0xN3

The system attempted to free some memory which it used for internal purposes, and ***freemem*** returned the error value N3. The system called ***freemem*** from address N2. The address which was being freed was N1. Caused by corrupted memory.

Corrupted MRdrivers

Produced at the same time as internal error 500, described above.

Booting from /..

Produced when a reset fails to find a suitable boot block on any drive.

Header checksum error

Error in a block device header checksum.

Found blocksize too large

Produced when tape read fails.

Device %s swapped, write still pending

Put the disk back in.

Block %d on %s is unreserved: Volume may be damaged

Run `fscheck` to correct potential file system damage.

Switch 2 open, using SA: for console

Console is serial port, only produced if disk co-processor card is not present, and DIP switch is set for serial port as console.

CON SA SB CENT NUL

List of available stream device names.

System closing pathname

A transient program left a file open when it terminated. The system closes this file and prints a warning message. The autoclosing of files was disabled by `option 7` in Version 3.

Index

Table of Contents