THIS MANUAL CONTAINS THE FOLLOWING SECTIONS -

* DEVICE DRIVERS
* I/O CONTROL CALLS
* BIOS CALLS
* APPENDICES
## CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver Summary</td>
<td>1</td>
</tr>
<tr>
<td>Uart Drivers</td>
<td>4</td>
</tr>
<tr>
<td>Clock Driver</td>
<td>6</td>
</tr>
<tr>
<td>Liquid Crystal Display Driver</td>
<td>9</td>
</tr>
<tr>
<td>Keyboard Driver</td>
<td>11</td>
</tr>
<tr>
<td>Parallel Port Driver</td>
<td>13</td>
</tr>
<tr>
<td>Video Driver</td>
<td>14</td>
</tr>
<tr>
<td>Rom Driver</td>
<td>16</td>
</tr>
<tr>
<td>Ram Driver</td>
<td>18</td>
</tr>
<tr>
<td>Power Down Driving</td>
<td>20</td>
</tr>
</tbody>
</table>
Device Driver Summary

I/O on the Magnum is normally performed through calls to the BIOS device drivers, either through Mdos or via the Bicall mechanism. This interface isolates the programmer from the hardware peculiarities of the machine on which he or she is working. This section of the manual provides a summary of the device drivers defined for the Magnum; detailed summaries of the individual devices follow, and should be read in conjunction with the Bicall documentation. This documentation assumes familiarity with Mdos Version 2 programming standards, and should be used with that operating system’s programmer’s reference manual.

At boot time the following character devices are defined to Mdos:

'CON' - Read/write; I/O control capable; console input; console output. Read calls to this device are vectored to the internal keyboard device 'KBD'; (see below); write calls are vectored to the liquid crystal display 'LCD'; (see below).

'AUX' - Read/write; I/O control capable. I/O to this device is performed to the first serial port (UART0). Input and output are buffered within the driver and flow control via the standard 'X-ON/X-OFF' protocol is supported.

'AUX2' - Read/write; I/O control capable. I/O to this device is performed to the second serial port (UART1). Input and output are buffered within the driver and flow control via the standard 'X-ON/X-OFF' protocol is supported.

'CLK' - Read/write; I/O capable. Reads from this device return the current time of day and date; writes update these parameters to new values. The format is as defined for the standard Mdos real-time clock device.

'LCC' - Write only; I/O capable. Writes to this device appear on the internal liquid crystal display. By default this device interprets the ANSI standard control sequences for display devices; within device limitations.

'KBD' - Read only. Reads from this device access the internal Magnum keyboard. The driver buffers the keyboard to allow user type-ahead. The function keys may be re-defined via a proprietary ANSI standard control sequence sent to an ANSI output device ie either 'LCD' or 'VID'.
PRN - Write only. Writes to this device are passed to the parallel output port, via a standard hardware handshake mechanism. Output to this device is not buffered.

VCON - Read/write; I/O control capable; character special device. Reads from this device are vectored to the internal keyboard device, 'KBD'; writes are vectored to the internal Magnum video device, 'VID'. This device is defined as the special character device to Msdos, enabling high speed output via the Interrupt 27 mechanism. This device is the usual alternate console.

VIC - Write only; I/O control capable. Writes to this device are sent to the internal Magnum video. By default this device interprets the Ansi standard control sequences for character devices, again within the limitations imposed by hardware.

In addition, the Magnum defines a number of block devices to Msdos. These are listed here. It should be noted that block devices under Msdos are not named but are referenced solely by drive letter, reflecting the order of their declaration to the operating system.

ROM - Write protected block device; 3 units defined. Unit 0 accesses the internal Magnum ROM area; of size 128K bytes. This unit is used to store the BIOS, Msdos itself, and a number of the utility programs. Units 1 and 2 access the left and right removable ROM modules respectively. These modules can each contain up to 128K bytes of data, although typically they contain less; they are usually used to store utility programs.

RAM - Read/write block device; 1 unit defined. This device accesses the internal ram area allocated to permanent storage; the size of this area is defined by the memory configuration in the machine at boot time. This device is usually restricted in size to 160K bytes.

Finally, systems based on the Magnum disk expansion box (MEB) provide a floppy disk driver. This has the following characteristics -

FLOPPY - Read/write block device; 2 units defined. The floppy hardware can support double sided, double density 5.25" disks; although other formats are allowed by the software
Each device supports the 13 standard Msdos interface calls for device drivers. I/O control calls are supported on an individual device basis, for special I/O purposes, such as screen size determination for the display devices. However, in addition to the standard Msdos interface, these calls, along with a number of others, may be accessed through the Bical mechanism. Bicalls are typically used where an I/O call would be unwieldy or inappropriate, such as the setting of alarms in the real time clock driver or interrogating driver version numbers. It is rare that a user would need to access the Bios through the Bical mechanism, as the Msdos interface is usually more appropriate and more portable.
Serial Port (UART) Driver

The UART driver controls both serial channels. It is declared to Msdos as two separate devices, 'AUX' and 'AUX2'. Internal Bios code maps accesses to these devices to the UART sub-system, units 0 and 1 respectively. For the purposes of this description we will consider only UART0; any information may be used identically for UART1.

The driver itself may conveniently be divided into two sections— one interrupt driven in response to hardware interrupts from the UART; the other invoked through Msdos or the Bicall mechanism. These two sections will be considered in turn.

The UART interrupt handler, 'uuint', is invoked in response to both transmit and receive interrupts. The transmitter will interrupt when its holding register becomes empty, as the character is transmitted; the receiver will interrupt as a character is received.

Consider first the transmit interrupt. The handler will take a character from the transmit buffer, 'txbuff' and write it to the UART. If the buffer is empty, then no character is transferred. If the character received is an 'X-OFF' the transmitter is de-activated, and no further characters will be sent until the receipt of an 'X-ON' character.

The receiver interrupt is processed in a similar fashion - the character is read from the UART and written into the receive buffer, 'rxbuff'. If this buffer fills to a set high-water mark (currently 10 characters from full) then the transmitter is activated and an 'X-OFF' character is transmitted. When the buffer subsequently drains to a set low-water mark (currently 10 characters from empty) the transmitter is activated again and an 'X-ON' character is sent. Currently, both 'rxbuff' and 'txbuff' are of size 80 characters.

Msdos and Bicalls transmit and receive characters through the routines 'uibwrite' and 'uibread', respectively. 'Uibwrite' transfers the given number of characters into 'txbuff'; if in this process 'txbuff' becomes full, the driver 'sleeps' the machine until space is available. Similarly, 'uibread' drains the requested number of characters from 'rxbuff'. If there are insufficient characters in the buffer to complete the transfer, the driver returns immediately with the transfer count field in the I/O request record set to reflect the actual number of characters that were read. There is however one important exception to this rule — if the transfer request was for only a single character, and 'rxbuff' is empty, then the machine 'sleeps' until a character becomes available.

The Msdos flush routines act as would be expected; 'Uiflush' empties 'rxbuff'; 'uioflush' empties 'txbuff'.
The I/O control read call is used to interrogate the number of characters stored in the rxbuffer.
Clock Device Driver

The clock sub-system implements the real-time clock facility supported by Msdos, with an extension to a powerful alarm capability. The basic structure of the driver is shown diagrammatically in the associated figure.

The clock is defined to Msdos as a read/write character device with I/O capability named 'CLK'; it is declared as the special CLOCK device, which means it will be interrogated by Msdos whenever the time and date are required.

'Ccbread' and 'Ccbwrite' implement the Msdos standard read and write time and date calls, as defined for the clock device. These routines simply interface to the hardware, converting between Msdos standard format and that expected by the clock hardware itself. Two points should be noted. The clock does not support time of day to an accuracy greater than one second; thus the hundredths-of-seconds field used by Msdos will always be '0' on a read, and is ignored on a write. Secondly, the clock cannot adequately support the Msdos concept of date; therefore this has been implemented in software; the day is maintained by the clock sub-system in a memory structure known as 'highcore', and is incremented each night at midnight via an internally maintained alarm. In all circumstances this should be invisible to the user.

I/O control calls are used in the clock solely to enable and disable three minute shutdown in the Magnum (see Power Down Modes section in this manual).

The clock hardware is initialised to interrupt at a pre-set time - the so called alarm capability. As the clock itself can only store one alarm at any time, the facility in the magnum to accommodate a number of alarms is implemented in software; via a table of pending alarms 'ccalarmtable'. The next pending alarm from this table is programmed into the clock; the clock interrupt handler processes the alarm and re-programmes the clock for the next pending.

Alarms are set and cleared by means of Bicalls; these are documented in that section of this manual. In summary however, alarms have the following characteristics -

- **time** - the time and day at which the alarm will trigger
- **type** - an alarm may be defined as one of several distinct types. These are
  - Terminating - the alarm will occur once, and then will be automatically deleted.
  - Daily - the alarm will occur at the same time each day; after each occurrence it will be re-scheduled for the next day
Hourly  - identical to the above, save
        that the repetition rate is
        every hour
Minutely - again, except minutely
Secondly - again, except secondly

beep  - if this field is non-zero, then when
       the alarm triggers the Magnum will beep
       repeatedly for around 10 seconds, in
       order to alert the user that an alarm
       has occurred. For safety reasons, it is
       not permissible to set this field for a
       repetitive (ie non-terminating) alarm.

flag pointer - this field contains a long (32 bit)
        pointer to a location in RAM. This
        location thus addressed should be set
        to the state 'PSLEEP' before the alarm
        is activated; if and only if it is in
        this state, the alarm interrupt handler
        will change its state to 'PWOKEN' when
        the alarm triggers. If its not in this
        state, then not only will the location
        not be altered but the alarm will be
        deleted from the table. This is a
        safety mechanism to prevent the
        corruption of memory if the program
        that originated the alarm has
        terminated.

Source ID - This field is not interpreted at all by
        the clock sub-system. It is used by
        the program simply to provide a means
        of identifying it's own alarms from
        those that may have been set by other
        programs. Certain values are used by
        the Bios and the Magnum diary utility;
        they should be avoided by other
        programs.

Usually, a program will set a noisy, terminating alarm, and
forget it. When the alarm sounds, the user will activate the
appropriate function to check status; the program can then, by
comparing its alarms with the time of day, respond appropriately,
usually with an explanatory message.

If a program wishes to suspend execution pending the arrival of
an alarm, the 'wait' Bicall in the clock sub-system can be used;
this function will 'sleep' the Magnum until some event occurs; an
event is any interrupt - it may be the arrival of the alarm, or
it may be some other event, such as a key stroke, or the arrival
of a character at the parallel port. Here, the flag-pointer
mechanism can be used to resolve the uncertainty. If the location
pointed at had been set to 'PSLEEP' before the call to 'wait',
then the arrival of the alarm would have caused its alteration to
'PWOKEN'; if it still in its original state, then the alarm has
not arrived, and 'wait' may be called again.
In order to avoid any possibility that the alarm may already have triggered between the setting of the alarm and the call to `wait`, the algorithm employed for 'wait' may return immediately if before the alarm has triggered. Thus, it is essential that the call to 'wait' is contained within the appropriate `while` loop; for example:

```c
flag = PSLERP;
setalarm(........
do
    wait(........
while (flag == PSLERP);
```
Liquid Crystal Display Driver

The liquid crystal display, being a non interrupt driven device is relatively simple in structure. The major components are illustrated in the associated figure.

The driver maintains a RAM based memory image of the current display, 'lcbuff'; 'lccopy' is used to copy the contents of this buffer to the display hardware controller.

The display is declared to Msdos as a character device with I/O control capability. The device name is 'LCD'. It should be noted that write calls to the device 'CON' will be vectored to the Lcd driver in a manner identical to writes to 'LCD'. 'Lcwrt' implements the standard Msdos block write call. The requested number of characters are copied from the given location to 'lcbuff', starting at the location specified by the current cursor position. The driver configuration structure 'lconfig' is interrogated to determine display format characteristics; this structure is the standard 'VICOMF' structure as described in the Appendices. If the driver is in Ansi mode (see below) then all characters are processed according to the standard Ansi control sequences (see Ansi section in this manual); if Ansi mode is disabled only the following control characters are recognised - carriage return, linefeed, backspace and horizontal tab.

I/O control write is used for a number of different purposes. These are as follows -

Cursor Position - the display cursor is moved to the specified position on the screen. This mechanism is disabled whilst Ansi mode is enabled.

Screen Dump - the specified number of characters are copied from the specified location direct to the display buffer, starting at the current cursor position. Absolutely no control character processing is done - the text must be in direct screen image form. This mechanism provides extremely fast screen access - the magnum word processor, for example, dumps a full screen of information after every key stroke using this mechanism. The cursor is unaffected.

Screen Dump Home- identical to 'Screen Dump', save that the characters are copied from the top left hand corner position of the screen (the 'home' position) regardless of the current position of the cursor. The cursor is unaffected.
Display Mode - device characteristics (Ansi capability, auto line-wrap, etc) may be read.

Screen Format - the size of the screen, ie the number of rows and columns on the display, may be read.

It should be noted that all of the above calls, in addition to the ANSI control sequences, work identically for the liquid crystal display and the video. In this way it is possible to implement utilities that work compatably on both display devices. The Magnum utilities are written with this in mind.

Like the video driver automatic line wrap and screen scrolling have been supported however cursor activity at the end of the screen is slightly different. After 80 characters have been displayed on any one line the cursor is not advanced to the next character position but is held under the last character. When the 81st character is sent it is displayed on the next line with the cursor in the 2nd character position. When the last line on the screen is reached and the 81st character is to be displayed the screen is scrolled.
Keyboard Device Driver

The keyboard driver may be divided into two sections - one interrupt driven and invoked upon each key stroke; the other invoked by Msdos or through the Bicall mechanism. These sections will be described in turn.

The keyboard hardware causes an interrupt which transfers control to the keyboard interrupt handler, 'kbin'. The scan code is read from the keyboard data register and mapped to its ASCII equivalent, via the lookup table 'kbmaptbl' and according to the state of the keyboard shift-lock flag (see below). The resultant character is placed in a first-in, first-out (FIFO) buffer, 'kbuff'. The following characters are treated specially by the interrupt handler and are not copied in the normal way -

CTRL '<' - decrease lcd screen contrast
CTRL '>' - increase lcd screen contrast
SHIFT CTRL '?' - toggle key click on and off
LOCK - toggle shift lock flag on and off
SHIFT CTRL 'S' - enable 3 minute shutdown
SHIFT CTRL 'C' - disable 3 minute shutdown
SHIFT CTRL 'P' - terminate 'sleep' in parallel port driver
SHIFT CTRL CHR - Any other shifted control key is ignored

The interrupt handler then terminates.

The keyboard ('KBD') is defined to Msdos as a read-only character device. The major call is the block read handler 'kbread'. Upon a read request for N characters, 'kbread' will copy between one and N characters from the buffer to the requested location. If the buffer is empty, the driver will 'sleep', ie put the machine into its partially powered down state, until a key is pressed and the interrupt handler asynchronously places a character in the buffer. At this point the 'sleep' will return and the read operation will return with one character. If there are less than N characters in the buffer, then all those present will be read as requested and the driver will return. As the characters are copied, any function key string mappings are performed. The function key lookup table 'kbfnmaptbl' is initialised at boot time to assign all function keys to their default values. New mappings are assigned via the Ansi sequence interpreter 'ankmap'.

'Kbdread' implements the non-destructive, non-wait read call used by Msdos for console status interrogation. The most recent character in 'kbuff', if any, is returned. It should be noted
that this call will always return immediately with the requested data - the machine will never be powered down. When it is desired that the machine 'sleep' on a keystoke, then the 'kbread' call should be used.

'Kbflush' flushes the keyboard buffer.
Parallel Port Driver

As no buffering of output characters is performed, the parallel port driver is simple in structure.

The parallel port is defined to Msdos as a write only character device, without I/O capability. The device name is 'PRN'.

The major call implements the block write function. Under hardware handshake control, the requested number of characters are written in turn to the parallel output port. While waiting for a handshake response the driver 'sleeps' the machine. The driver will not return until all requested characters have been transferred, or until a printer abort request is lodged. This latter is generated by the keyboard in response to a special control sequence (see Keyboard driver documentation).
Video Driver

The internal video on the Magnum is based around the Intel i876 CRT controller and the internal i86 DMA controllers.

The i8276 requests characters for display on a line by line basis as it is displaying one character line, it reads into its buffer the 80 characters for the next character line. The transfer of characters from RAM to the i8276 is performed by DMA; this transfer is performed for a complete screen without processor intervention, via normal hardware synchronisation between the DMA controller (DMAC) and the i8276. After a complete screen has been transferred, the DMAC interrupts the processor, which re-initialises it for the next frame. In this way interrupts from the i8276 may be disabled when the screen is static.

Scrolling is done by changing the relative synchronisation between the top of the video screen and the beginning of the video buffer. This is performed by the video and DMA interrupt handlers during the vertical retrace period, and leads to fast, stable scrolling.

The video driver is declared to MSDOS as a write only character device with I/O control capability, 'VID'. It should be noted that write calls to the device 'VCON' will be vectored to this driver. Standard character writes are performed in the normal fashion by the block write routine, 'vibwrite'. The requested number of characters are transferred to the screen. If the device is in ANSI mode (see below) then all characters are processed according to the standard ANSI control sequences (see ANSI section in this manual); if ANSI mode is disabled only the following control characters are recognised - carriage return, linefeed, backspace and horizontal tab.

I/O control is used for a number of different purposes. These are as follows -

Cursor Position - the video cursor is moved to the specified position on the screen. This mechanism is disabled whilst ANSI mode is enabled.

Screen Dump - the specified number of characters are copied from the specified location direct to the video buffer, starting at the current cursor position. Absolutely no control character processing is done - the text must be in direct screen image form. This mechanism provides extremely fast screen access - the magnum word processor, for example, dumps a full screen of information after every key stroke using this mechanism. The cursor is unaffected.
Screen Dump Home—identical to 'Screen Dump', save that the characters are copied from the top left hand corner position of the screen (the 'home' position) regardless of the current position of the cursor. The cursor is unaffected.

Video Mode — device characteristics (Ansi capability, auto line-wrap, etc) may be read.

Screen Format — the size of the screen, ie the number of rows and columns on the display, may be read.

It should be noted that all of the above calls, in addition to the Ansi control sequences, work identically for the liquid crystal display and the video. In this way it is possible to implement utilities that work compatably on both display devices. The Magnum utilities are written in this way.

Unlike the liquid crystal display, the video hardware is capable of text highlighting. Attributes supported are —

- **Blink** — the text will flash
- **Bold** — the text will be of higher intensity
- **Uline** — the text will be underlined
- **Reverse** — the text will be displayed in inverse video

Any number of the above attributes may be combined. An attribute mode is enabled by writing a control string to the driver (see Ansi documentation). That attribute will then remain in force until the next attribute mode is specified, or the end of screen, whichever is closer. Each attribute mode consumes one character position on the screen; this is known as 'visible attribute' capability.

It is much more desirable to use the Ansi control sequences to control attributes, rather than programming the device directly via a 'Screen Dump' call. The speed penalty is minimal, and the sequences will not behave strangely on devices other than the video, such as the liquid crystal display.
ROM Device Driver

The ROM driver imposes onto areas of memory occupied by ROM the format required by Msdos for block devices. Information may then be stored in the machine in a form accessible by the operating system in the normal way.

The ROM drive is declared to Msdos as a standard block device, comprising three units. As this device is declared before any other block devices, it occupies drives 'A', 'B', and 'C' in the Magnum. Drive 'A' accesses the internal 128K ROM space; Drives 'B' and 'C' access the removeable ROM modules. These latter can contain up to 128k bytes each, although typically they will contain less. The standard modules shipped with the Magnum typically contain 32k bytes each.

The minimum active functions required by Msdos for a block device are 'Block Read/Write', 'Read BPB', and 'Read Media Code'. These will be discussed in turn.

As shown on the associated diagram, the block read/write function, 'rbread', is the most complex of the functions. Consider first accesses to the internal ROM device, drive 'A'. Msdos requests a transfer of a number of sectors, starting at a given start sector. This sector number is mapped to a physical memory address, according to the ROM drive format definition (see below). Data is copied from that address to the given buffer.

Accesses to either of the removeable ROM modules are slightly more complex. Due to limitations on the address space of the i186 processor, both ROM modules share the same physical address. Only one module may be selected at any one time, by means of the ROM control location 'rocontrol'. Depending on the unit number specified by Msdos, the appropriate ROM module is selected, and the data transfer proceeds as described for the internal ROM device.

Only read accesses are valid for these drives; any attempt to write the device will provoke a 'Write Protect' error. Similarly, accesses to the removeable ROMs are preceded by an integrity check on the specified module; if data inappropriate to a ROM device is present, then a 'Not Ready' error is returned.

The 'Get BPB' call is used by Msdos to quantify the device characteristics — it specifies such things as sector count and sector size. 'Rogbpbb' implements this call for the ROM drives, using the ROM configuration structure 'roconfig'. In the case of the removeable modules there is a possibility that this structure is out of date. For this reason, 'romod' is invoked to re-calculate the configuration from the information stored in the ROM module itself.

'romcode' implements the Msdos call to read the media descriptor code for that device. In the ROM drive this call simply returns a constant value appropriate to the device.
The ROM drive memory format will now be described. It should be noted at this point that this information is included for general informational purposes only. It is not intended, nor will it be supported, that users should access the data in the ROMs other than through the ROM device driver. Those users who wish to program ROMs in the format appropriate to these drives should contact a Dulmont representative.

ROM sectors are typically of size 128 bytes, and start at the lowest physical address in the ROM, and proceed sequentially upwards. Thus sector 0 in the internal ROM drive has physical address 'E0000'; sector 1 will have address 'E0080'; sector 2 will have address 'E0100'; and so on. There is usually one reserved sector; this is used by the ROM driver for the integrity checking described above. Sectors are contiguous i.e. there is no data stored between them; the Bios and some of the utilities use this facility to execute directly from their ROM file image. The size of the FATs and directories is determined on a per-device basis, as appropriate for the information to be stored.
RAM Device Driver

The RAM driver imposes onto areas of memory occupied by RAM the format required by Msdos for block devices. Information may then be stored in the machine in a form accessible by the operating system in the normal way.

The RAM drive is declared to Msdos as a standard block device, comprising one unit. As this device is declared immediately after the ROM driver, it occupies drive 'D' in the Magnum.

The amount of memory allocated to the RAM drive is determined at boot time by the memory configuration of the Magnum. The RAM drive memory area begins at the top of the internal RAM address space, i.e. '3fff', and proceeds downwards towards physical address '00000' i.e in the opposite direction to the ROM drive. Sector numbers proceed in the same direction i.e sector 0 begins at location '3ff80'; sector 1 begins at '3ff00'; and so on. System memory, i.e that used by Msdos and the Bios for normal transient store, begins at '0000' and proceeds upwards. Both these areas are sized at boot time; system memory by the Bios kernel, and RAM drive memory by 'rsinit'. In most machines these two areas are distinct i.e separated by a gap in the memory space where there is no RAM. In a machine which has 256K of internal RAM, there is no gap. In these circumstances, an arbitrary division must be made. The current allocation is 96K for system memory, 160K for RAM disk; this arrangement may however alter in the future.

The minimum active functions required by Msdos for a block device are 'Block Read/Write', 'Read BPB', and 'Read Media Code'. These will be discussed in turn.

As shown on the associated diagram, the block read/write function is implemented by 'rsbreadwrite'. Msdos requests a transfer of a number of sectors, starting at a given start sector. This sector number is mapped to a physical memory address, according to the drive format described above. Data is copied between that address and the given buffer.

If the serial number of your Magnum exceeds '01400' then it supports hardware protection for the upper 128K of RAM i.e. addresses '20000' to '3fff'. Read/write access to these locations is guarded by a RAM protect latch, 'rsprotect'. If memory allocated to transient storage exceeds 128K, then the protection must be disabled, in order to allow unimpeded access to upper memory. If, however, as is normally the case, there is less than 128K of transient storage, then upper memory is guarded, and access only enabled within the RAM driver itself.

The 'Get BPB' call is used by Msdos to quantify the device characteristics – it specifies such things as sector count and sector size. 'Rsgbpb' implements this call for the RAM drive, using the RAM configuration structure 'rsconfig' calculated by 'rsinit' at boot time.
'Rsmcode' implements the Msdos call to read the media descriptor code for that device. In the RAM drive this call simply returns an arbitrary but constant value appropriate to the device.

The RAM drive memory format will now be described. It should be noted at this point that this information is included for general informational purposes only. It is not intended, nor will it be supported, that users should access the data in the RAMs other than through the RAM device driver.

RAM sectors are typically of size 128 bytes, with 1 sector per cluster. There are no reserved sectors. Sectors are contiguous i.e. there is no data stocked between them. There is one FAT and 32 directory entries: the size of the FATs and the number of sectors is determined at boot time by the size of the RAM device.
Power Down Modes

There are 4 major power supplies in the Dulmont Magnum; these are:

- Memory power: used to power the CMOS static ram and the real time clock. This supply is always enabled, even when the machine is off.

- Base power: used to power both the internal and the plug in roms, the liquid crystal display, as well as much of the support circuitry. This supply is enabled whenever the liquid crystal display is alive.

- CPU power: used to power the CPU and its immediate support circuitry. This supply is enabled whenever processing is in progress.

- Accessory power: used to power the internal video generator and the expansion bus. Enabled only when the video is running or an expansion unit is plugged in.

The following figures represent the approximate power consumption figures in the Dulmont Magnum:

- Power consumption with Memory, Base, CPU and Accessory: 2.0 amps
- Power consumption with Memory, Base and CPU: 1.4 amps
- Power consumption with Memory, Base: 0.2 amps
- Power consumption with Memory: 0.0 amps

Power capacity in internal batteries: 4.0 Ahrs

From these figures it is clear that a Magnum running with CPU enabled, even without video, will drain the batteries in a short time. For this reason, power reduction by selective shutdown is a critically important part of the Magnum design. This section will give a broad overview of the mechanisms employed, to the extent necessary for the programming of applications to be used in portable mode.

For reasons which will be described below, the following discussion concerns a machine running with video disabled and no expansion unit in place.

The Magnum Bios powers down the CPU whenever the machine is waiting for some event to occur. Usually, the event will be the completion of some I/O operation, such as a keystroke on the internal keyboard, or the transmission of a character through the serial or parallel ports. In hardware terms, an event is simply the occurrence of some interrupt; thus here follows a complete list of event sources in the Magnum:
Version 1.0X  POWER DOWN MODES

Keyboard  - The keyboard will generate an interrupt upon detection of any keystroke.

Serial Ports  - Either serial port (UART) will generate an interrupt after the transmission or reception of any character

Parallel Port  - Completion of an input or output cycle through the parallel port will generate an interrupt.

Real Time Clock  - The clock can interrupt either at a set frequency, or at a pre-set time (an alarm); normally, only the latter mode is enabled.

ROM Modules  - In order to allow use of the ROM module ports for device expansion, interrupt lines are provided at both interfaces. When these ports are used for ROM modules, these lines are inactive.

3 min S/down  - 3 minutes after the CPU is turned off, if Base power is enabled, an interrupt is generated. This interrupt is used to implement the 3 minute shutdown feature (see below).

Drivers which cannot complete an I/O request disable CPU power pending the appropriate event; this operation is referred to in the driver documentation as 'sleeping' the machine. In order to explain, consider a request to the keyboard driver for a character. If there is a character in its buffer, the I/O can complete immediately. If, however, the keyboard buffer is empty, then the I/O cannot complete until a key is pressed. The driver 'sleeps' the machine – CPU power is removed whilst base power remains enabled. As it is Base power that maintains the display, the user will not be aware that the machine has powered down in any way.

If a key is now pressed, the hardware restores power to the CPU; the keyboard interrupt will be recognised and processed by the keyboard interrupt handler, which places the appropriate character in the keyboard buffer. The interrupt then terminates, returning control to the program that was running when operations were suspended – in this case, the keyboard driver. This driver checks the buffer, and, finding that there is now a character present, the I/O operation is completed.

If the interrupt had been from some other source, however, such as the reception of a character by the parallel port, then the keyboard driver would find upon regaining control that its buffer was still empty; it would thus re-sleep the machine, until, eventually a character arrived.
This fundamental operation is repeated in concept throughout the Magnum I/O system. The serial port drivers, for example, will sleep the machine in between character transmissions; the CPU is enabled only long enough to process the interrupt and pass another character to the UART. In this way, average power consumption in the Magnum under normal usage is reduced to a small fraction of its peak value.

This approach imposes an important restriction on programmers working with the Magnum — programmed loops on pending I/O must be avoided. This is not difficult to achieve under the calls available in Msdos Version 2; however, it implies that the venerable ‘constat’ spin loop common to CP/M and Msdos Version 1 is no longer appropriate.

The internal video generator in the Magnum makes use of the DMA controller present in the i866; for this reason, coupled with the fact that the video generator must be powered continuously to maintain the display, the Magnum cannot power down when the video display is enabled. The Magnum video should not be used with battery power; the consumption is simply too great. When video is enabled, the ‘sleep’ mechanism is disabled by the Bios software; the same calls are issued but the CPU will loop within the Bios awaiting the completion of the specified I/O.

A feature is provided in the Magnum to completely shut the machine down after 3 minutes of inactivity. The function comes into effect after the machine has been put to ‘sleep’ in the manner described above — ie CPU power disabled with Base power enabled. An internal counter started by the switching off of the CPU power generates an interrupt after 3 minutes; the interrupt handler then disables all power supplies, thus completely switching off the machine in a manner identical to the pressing of the ‘OFF’ key.

When the ‘ON’ key is pressed, CPU and base power are restored by the hardware; the Bios restores the machine state precisely to that present before the arrival of the shutdown interrupt — the LCD screen is restored, as are the serial port characteristics and all other machine variables. Control then returns to the previous task, usually one of the drivers awaiting completion of I/O in the manner described above. As that I/O will still not have completed (no events can occur whilst the machine is switched completely off), the driver will re-‘sleep’ the machine — disabling CPU power but retaining Base power. If the I/O has still not completed after a further three minutes, then the shutdown sequence is again performed.

Three minute shutdown eliminates the problem of an unattended machine, running without mains power, with Base power draining the batteries past their safe level. There are however circumstances where shutdown is not desirable — such as, for example, an unattended Magnum coupled to a modem awaiting communications. For this reason, the facility may be disabled —
both from the keyboard (see the keyboard driver documentation), and under programmer control (see the Bicall documentation for the Clock Sub-system). This facility should clearly be treated with care, as damage to the batteries and internal data may occur if the batteries are exhausted. Three minute shutdown will always be re-enabled immediately after the machine has been switched on via the 'ON' key.
<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Driver</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Ioctl Fn</td>
<td></td>
</tr>
<tr>
<td>Aux</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Read No. chars on queue 4</td>
</tr>
<tr>
<td>Aux2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Read No. chars on queue 6</td>
</tr>
<tr>
<td>Clock</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Read Shutdown Mode 8</td>
</tr>
<tr>
<td></td>
<td>Write Shutdown Mode 9</td>
</tr>
<tr>
<td>Keyboard/Lcd</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Read Vidmod 11</td>
</tr>
<tr>
<td></td>
<td>Read from screen home 12</td>
</tr>
<tr>
<td></td>
<td>Get Cursor Position 13</td>
</tr>
<tr>
<td></td>
<td>Read Screen Format 14</td>
</tr>
<tr>
<td></td>
<td>Read Screen from cursor 15</td>
</tr>
<tr>
<td></td>
<td>Write to screen home 16</td>
</tr>
<tr>
<td></td>
<td>Set Cursor Position 17</td>
</tr>
<tr>
<td></td>
<td>Write to screen from cursor 18</td>
</tr>
<tr>
<td>Keyboard/Video</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Read Vidmod 20</td>
</tr>
<tr>
<td></td>
<td>Read from screen home 21</td>
</tr>
<tr>
<td></td>
<td>Get Cursor Position 22</td>
</tr>
<tr>
<td></td>
<td>Read Screen Format 23</td>
</tr>
<tr>
<td></td>
<td>Read screen from cursor 24</td>
</tr>
<tr>
<td></td>
<td>Write to screen from home 25</td>
</tr>
<tr>
<td></td>
<td>Set Cursor Position 26</td>
</tr>
<tr>
<td></td>
<td>Write to screen from cursor 27</td>
</tr>
</tbody>
</table>
## CONTENTS

### IOCTL V1.0

<table>
<thead>
<tr>
<th>Driver</th>
<th>_IOCTL Fn</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCD</td>
<td>Read Vidmod</td>
<td>26</td>
</tr>
<tr>
<td>LCD</td>
<td>Read screen from home</td>
<td>29</td>
</tr>
<tr>
<td>LCD</td>
<td>Get Cursor Position</td>
<td>30</td>
</tr>
<tr>
<td>LCD</td>
<td>Read Screen Format</td>
<td>31</td>
</tr>
<tr>
<td>LCD</td>
<td>Read screen from cursor</td>
<td>32</td>
</tr>
<tr>
<td>LCD</td>
<td>Write to screen from home</td>
<td>33</td>
</tr>
<tr>
<td>LCD</td>
<td>Set Cursor Position</td>
<td>34</td>
</tr>
<tr>
<td>LCD</td>
<td>Write to screen from cursor</td>
<td>35</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Read Vidmod</td>
<td>36</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Read screen from home</td>
<td>37</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Get Cursor Position</td>
<td>38</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Read Screen Format</td>
<td>39</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Read screen from cursor</td>
<td>40</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Write to screen from home</td>
<td>41</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Set Cursor Position</td>
<td>42</td>
</tr>
<tr>
<td>VIDEO</td>
<td>Write to screen from cursor</td>
<td>43</td>
</tr>
</tbody>
</table>
INTRODUCTION

This chapter describes the Dulmont Magnum's usage of the MSDOS I/O control (ioctl) function documented in the Programmer's Reference Manual, page 1-121.

The ioctl function reads or writes information to or from an open device handle. The ioctl call is routed to the correct service routine via a three level vectoring system based on a device handle, an ioctl function (either read or write) and a command offset in a parameter block - the address and size of which are set up in the calling registers. All the commands supported by the Dulmont Magnum are outlined in Table 1.

An ioctl call is invoked through interrupt INT 21. Parameters are passed to and from the service routine via registers and a parameter block set up before the interrupt is called. The call conventions are:

```
ah = 44H (ioctl function code)
bx = handle from previous open on device
bl = 0,1,2.. (drive for FD function calls)
dsiex = address of parameter block
cx = 14 (size of parameter block)
al = 2,3,4,5 (function code)
```

Parameter block

```
struc
    cmd   dw (?); command
    troff dw (?); address of buffer or struc
    tcrnt dw (?); size of buffer or struc
    filler dd (?); reserved
ends
```

The results of the service routine are returned by either updating the parameter block and/or the ax register.

To perform an IOCTL function on a device it must firstly be opened. This is done as described in the MSDOS PROGRAMMER'S REFERENCE MANUAL.
INTRODUCTION

Valid device driver names are listed below:

<table>
<thead>
<tr>
<th>Device Name</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUX</td>
<td>Serial Aux Driver</td>
</tr>
<tr>
<td>AUX2</td>
<td>Serial Aux2 Driver</td>
</tr>
<tr>
<td>CLK</td>
<td>Clock Driver</td>
</tr>
<tr>
<td>FD</td>
<td>Floppy Disk Driver</td>
</tr>
<tr>
<td>KBD</td>
<td>Keyboard driver</td>
</tr>
<tr>
<td>CON</td>
<td>Keyboard, LCD Driver</td>
</tr>
<tr>
<td>LCD</td>
<td>LCD driver</td>
</tr>
<tr>
<td>VCON</td>
<td>Video, Keyboard Driver</td>
</tr>
<tr>
<td>VID</td>
<td>Video Driver</td>
</tr>
</tbody>
</table>
**AUX SUBSYSTEM**

Supported IOCTL Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ IOCTL</td>
<td>0</td>
</tr>
<tr>
<td>Read NO. chars on queue</td>
<td>0</td>
</tr>
</tbody>
</table>
AUX SUBSYSTEM

SUBSYSTEM: AUX

Name(fn): Ioctl Read - Read Number of Queued Characters

Call
ah = 44H (Ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "AUX" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struct
  cmd
dw (?) ; = 0
troff
dw (?) ; = address of word 'count'
trcnt
dw (?) ; = 2 (size of word 'count')
filler
dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)
Carry not set:
ax = 14 (size of parameter block in bytes)
count = current number of characters in the queue

Explanation
Determine the number of characters in the receiver character queue for the AUX device.

On exit the value of the word 'count' is set to the number of characters in the AUX receive queue.
AUX2 SUBSYSTEM

Supported IOCTL calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ IOCTL</td>
<td>0</td>
</tr>
<tr>
<td>Read NO. chars on queue</td>
<td>0</td>
</tr>
</tbody>
</table>
AUX2 SUBSYSTEM

SUBSYSTEM: AUX2

Name(fn): IOCTL Read - Read Number of Queued Characters

Call

ah = 44H (IOCTL fn number)
a1 = 2 (function code)
bx = handle (from OPEN on "AUX2" device)
ds:dx = parameter block address

cx = 14 (size of parameter block in bytes)

Parameter Block

struct

    cmd          dw (?) ; = 0
    troff        dw (?) ; = address of word 'count'
    trcnt        dw (?) ; = 2 (size of word 'count')
    filler       dw 4 dup (?); reserved

ends

Return

Carry set:

ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)

Carry not set:

ax = 14 (size of parameter block in bytes)
count = current number of characters in the queue

Explanation

Determines the number of characters in the receiver character queue for the AUX2 device.

On exit the value of the word 'count' is set to the number of characters in the AUX2 receive queue.
**CLOCK SUBSYSTEM**

**Supported Ioctl Cells**

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ IOCTL</td>
<td>55</td>
</tr>
<tr>
<td>Set Shutdown Mode</td>
<td></td>
</tr>
<tr>
<td>WRITE IOCTL</td>
<td>55</td>
</tr>
<tr>
<td>Read Shutdown Mode</td>
<td></td>
</tr>
</tbody>
</table>
CLOCK SUBSYSTEM

SUBSYSTEM: CLOCK

Name(fn): Ioctl Read - Read Shutdown Mode

Call
ah = 44H (Ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "CLK" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struct
cmd
dw (?) ; = 55
troff
dw (?) ; = address of byte 'mode'
trcnt
dw (?) ; = 1 (size of byte 'mode')
filler
dw 4 dup (?); reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)

Carry not set:
ax = 1 (size of byte 'mode')
mode = 0 if 3 minute shutdown disabled
     = 1 if 3 minute shutdown enabled

Explanation

Sets the value in the byte 'mode' to show whether 3 minute shutdown is enabled or not.
CLOCK SUBSYSTEM

SUBSYSTEM: CLOCK

Name(fn): Ioclt Write - Set Shutdown Mode

Call
ah = 44H (Ioclt fn number)
al = 3 (function code)
bx = handle (from OPEN on "CLK" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struc

    cmd      dw (?) ; = 55
    troff    dw (?) ; = address of byte 'mode'
    trcnt    dw (?) ; = 1 (size of byte 'mode')
    filler   .dw 4 dup (?); reserved

ends

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation

Reads the value in the byte 'mode' and enables or disables 3 minute shutdown appropriately.

On entry the value in the byte 'mode' is set as follows
mode = 0 if 3 minute shutdown is to be disabled
    1 if 3 minute shutdown is to be enabled
Supported ioctl Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ IOCTL</td>
<td></td>
</tr>
<tr>
<td>Read Vidmod</td>
<td>0</td>
</tr>
<tr>
<td>Read screen from home</td>
<td>1</td>
</tr>
<tr>
<td>Get Cursor Position</td>
<td>2</td>
</tr>
<tr>
<td>Read Screen Format</td>
<td>3</td>
</tr>
<tr>
<td>Read screen from cursor</td>
<td>4</td>
</tr>
<tr>
<td>WRITE IOCTL</td>
<td></td>
</tr>
<tr>
<td>Write to screen from home</td>
<td>1</td>
</tr>
<tr>
<td>Set Cursor Position</td>
<td>2</td>
</tr>
<tr>
<td>Write to screen from cursor</td>
<td>4</td>
</tr>
</tbody>
</table>
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(fn): Ioctl Read - Read Vidmod

Call
ah = 44H (Ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "CON" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struc
cmd
dw (?) ; = 0
troff
dw (?) ; = address of VIMOD structure
trcnt
dw (?) ; = 19 (size of VIMOD struct)
filler
dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block)

Explanation
Copies current LCD VIMOD structure (see Appendix C) into the address specified by troff in the parameter block.
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(fn): Ioctl Read - Read screen from home

Call
ah = 44H (Ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "CON" device)
cs:cx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struct
  cmd      dw (?) ; = 1
  troff    dw (?) ; = address of transfer buffer
  trcnt    dw (?) ; = no of bytes to be transferred
  filler   dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)
Carry not set:
ax = 14 size of parameter block in bytes

Explanation
Trcnt bytes are read into the transfer buffer from the console starting from the home position.
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(func): Ioctl Read - Get Cursor Position

Call
ah = 44H  (Ioctl fn number)
al = 2  (function code)
bx = handle  (from OPEN on "CON" device)
ds:dx = parameter block address
cl = 14  (size of parameter block in bytes)

Parameter Block
struc
cmd  db (?) ; = 2
troff  dw (?) ; = address of CPOS structure
trcnt  dw (?) ; = size of CPOS structure
filler  dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
    = 1  (invalid function)
    = 13 (invalid data)
    = 5  (access denied)

Carry not set:
ax = size of parameter block

Explanation
The current CPOS structure (see Appendix C) is copied into the buffer at the address specified by 'troff' in the parameter block.
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(fn): IOCTL Read - Read Screen Format

Call

ah = 44H  (IOCTL fn number)
al = 2  (function code)
bx = handle  (from OPEN on "CON" device)
ds:dx = parameter block address

cx = 14  (size of parameter block in bytes)

Parameter Block

struc
    cmd
    troff
dw (?) ; = 3
dw (?) ; = address of SFRMT struc
tcnt
dw (?) ; = 4  (size of SFRMT struc)
    filler
dw 4 dup (?) ;reserved
ends

Return

Carry set:

ax = 6  (invalid handle)
= 1  (invalid function)
= 13  (invalid data)
= 5  (access denied)

Carry not set:

ax = 14  (size of parameter block in bytes)

Explanation

The SFRMT structure (see Appendix C) is copied into buffer at the address specified by troff in the parameter block.
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(fn): Ioctl Read - Read screen from cursor

Call

ah = 44H  (Ioctl fn number)
al = 2     (function code)
bx = handle (from OPEN on "CON" device)
ds:dx = parameter block address
cx = 14   (size of parameter block in bytes)

Parameter Block
struc
    cmd      dw (?) ; = 4
    troff    dw (?) ; = address of transfer buffer
    trcnt    dw (?) ; = no of bytes to be transfered
    filler   dw 4 dup (?) ;reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
   = 1  (invalid function)
   = 13 (invalid data)
   = 5  (access denied)
Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
Trcnt bytes are read into the transfer buffer from the console starting at the current cursor position.
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(fn): Iocntl Write - Write to screen from home

Call
ah = 44H  (Iocntl fn number)
al = 3     (function code)
bx = handle (from OPEN on "CON" device)
ds:dx = parameter block address
cx = 14   (size of parameter block in bytes)

Parameter Block
struc
cmd     dw (?) ; = 1
troff   dw (?) ; = address of transfer buffer
trcnt   dw (?) ; = no. of bytes to be transfered
filler  dw 4 dup (?) ;reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
   = 1  (invalid function)
   = 13 (invalid data)
   = 5  (access denied)
Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the console starting at the home position. The cursor position is not affected.
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(fn): Ioctl Write - Set Cursor Position

Call

\[
\begin{align*}
ah & = 44H & \text{(Ioctl fn number)} \\
al & = 3 & \text{(function code)} \\
bx & = \text{handle} & \text{(from CFEN on "CON" device)} \\
ds:dx & = \text{parameter block address} \\
cx & = 14 & \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

```
struc
  cmd
  troff
  trcnt
  filler
end struc
```

Return

Carry set:

\[
\begin{align*}
ax & = 6 & \text{(invalid handle)} \\
& = 1 & \text{(invalid function)} \\
& = 13 & \text{(invalid data)} \\
& = 5 & \text{(access denied)}
\end{align*}
\]

Carry not set:

\[
ax = 14 & \quad \text{(size of parameter block in bytes)}
\]

Explanation

Sets the cursor position to the value set up in the CPOS structure (see Appendix C).

Note. This IOCTL call will not be put into effect if the ANSI field in the VIMOD structure is active.
KEYBOARD/LCD SUBSYSTEM

SUBSYSTEM: KEYBOARD/LCD

Name(fn): Ioctl Write - Write to screen from cursor

Call
    ah = 44H    (Ioctl fn number)
    al = 3      (function code)
    bx = handle (from OPEN on "CON" device)
    ds:dx = parameter block address
    cx = 14     (size of parameter block in bytes)

Parameter Block
struct
    cmd    dw (?) ; = 4
    troff  dw (?) ; = address of transfer buffer
    trcnt  dw (?) ; = no of bytes to be transferred
    filler dw 4 dup (?) ; reserved
ends

Return

    Carry set:
    ax = 6  (invalid handle)
    = 1   (invalid function)
    = 13  (invalid data)
    = 5   (access denied)
    Carry not set:
    ax = 14  (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the screen starting at the current cursor position.
SUBSYSTEM: LCD

Name(fn): Ioctl Read - Read screen from home

Call
aha = 44H  (Ioctl fn number)
al = 2    (function code)
bx = handle (from OPEN on "LCD" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block

  struct
  cmd    dw (?) ; = 1
  troff  dw (?) ; = address of transfer buffer
  trcnt  dw (? ; = no of bytes to be transferred
  filler dw 4 dup (?) ; reserved
  ends

Return

  Carry set:
  ax = 6  (invalid handle)
  = 1    (invalid function)
  = 13   (invalid data)
  = 5    (access denied)

  Carry not set:
  ax = 14 (size of parameter block in bytes)

Explanation

Trcnt bytes are read into the transfer buffer from the screen starting the home position.
SUBSYSTEM: LCD

Name(fn): Ioctl Read - Get Cursor Position

Call
ah = 44H  (Ioctl fn number)
al = 2  (function code)
bx = handle  (from OPEN on "LCD" device)
ds:dx = parameter block address
cx = 14  (size of parameter block in bytes)

Parameter Block
struct
  cmd  db (?) ; = 2
  troff  dw (?) ; = address of CPOS structure
  trcnt  dw (?) ; = size of CPOS structure
  filler  dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
     = 1  (invalid function)
     = 13 (invalid data)
     = 5  (access denied)
Carry not set:
ax = 14  (size of parameter block in bytes)

Explanation
The CPOS structure (see Appendix C) is copied into the buffer at the address specified by 'troff' in the parameter block.
SUBSYSTEM: LCD

Name(fn): Ioctl Read - Read Screen Format

Call
ah = 44H  (Ioctl fn number)
al = 2     (function code)
bx = handle  (from OPEN on "LCD" device)
ds:dx = parameter block address
cx = 14  (size of parameter block in bytes)

Parameter Block
struc
cmd        dw (?) ; = 3
troff      dw (?) ; = address of SFRMT struc
trcnt      dw (?) ; = 4  (size of SFRMT struc)
filler     dw 4 dup (?) ; reserved
ends

Return

 Carry set:
aX = 6  (invalid handle)
    = 1  (invalid function)
    = 13 (invalid data)
    = 5  (access denied)
 Carry not set:
aX = 14  (size of parameter block in bytes)

Explanation
The SFRMT structure (see Appendix C) is copied into the buffer at the address specified by troff in the parameter block.
SUBSYSTEM: LCD

Name(fn): Ioctl Read - Read screen from cursor

Call
ah = 44H  (Ioctl fn number)
al = 2     (function code)
bx = handle (from OPEN on "LCD" device)
ds:dx = parameter block address

cx = 14   (size of parameter block in bytes)

Parameter Block
struct
    cmd     dw (?) ;  = 4
    troff   dw (?) ;  = address of transfer buffer
    trcnt   dw (?) ;  = no of bytes to be transferred
    filler  dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
       1  (invalid function)
       13 (invalid data)
       5  (access denied)
Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
Trcnt bytes are copied into the transfer buffer from the screen starting at the current cursor position.
SUBSYSTEM: LCD

Name(fn): Ioctl Write - Write to screen from home

Call
ah = 44H  (Ioctl fn number)
a1 = 3    (function code)
bx = handle  (from OPEN on "LCD" device)
ds:dx = parameter block address
cx = 14   (size of parameter block in bytes)

Parameter Block
struct
    cmd      dw (?) ; = 1
    troff    dw (?) ; = address of transfer buffer
    trcnt    dw (?) ; = no. of bytes to be transferred
    filler   dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
      = 1  (invalid function)
      = 13 (invalid data)
      = 5  (access denied)

Carry not set:
ax = 14  (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the screen starting at the home position.
SUBSYSTEM: LCD

Name(fn): Ioctl Write - Set Cursor Position

Call

\[ \begin{align*}
ah &= 44\text{H} & \text{(Ioctl fn number)} \\
al &= 3 & \text{(function code)} \\
bx &= \text{handle} & \text{(from OPEN on "LCD" device)} \\
ds:dx &= \text{parameter block address} \\
\text{cx} &= 14 & \text{(size of parameter block in bytes)}
\end{align*} \]

Parameter Block

\[ \text{struc} \]
\[ \begin{align*}
\text{cmd} &= \text{dw (?) ; } = 2 \\
\text{tloff} &= \text{dw (?) ; } = \text{address of CP0S structure} \\
\text{trcnt} &= \text{dw (?) ; } = \text{size of CP0S structure} \\
\text{filler} &= \text{dw 4 dup (?) ; reserved}
\end{align*} \]
\[ \text{ends} \]

Return

Carry set:

\[ \begin{align*}
ax &= 6 & \text{(invalid handle)} \\
&= 1 & \text{(invalid function)} \\
&= 13 & \text{(invalid data)} \\
&= 5 & \text{(access denied)}
\end{align*} \]

Carry not set:

\[ \begin{align*}
ax &= 14 & \text{(size of parameter block in bytes)}
\end{align*} \]

Explanation

Sets the cursor position to the value set up in the CP0S structure (see Appendix C).

Note. This IOCTL call will not be put into effect if the ANSI field of the VIMOD structure is active.
SUBSYSTEM: LCD

Name(fn): Ioctl Write – Write to screen from cursor

Call
ah = 44H (Ioctl fn number)
al = 3 (function code)
bx = handle (from OPEN on "LCD" device)

ds:dx = parameter block address

cx = 14 (size of parameter block in bytes)

Parameter Block
struc

    cmd    dw (?) ; = 4
    troff  dw (?) ; = address of transfer buffer
    trcnt  dw (?) ; = no of bytes to be transferred
    filler dw 4 dup (?) ;reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the screen starting at the current cursor position.
## Supported IOCTL Calls

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ IOCTL</td>
<td></td>
</tr>
<tr>
<td>Read Vidmod</td>
<td>0</td>
</tr>
<tr>
<td>Read screen from home</td>
<td>1</td>
</tr>
<tr>
<td>Get Cursor Position</td>
<td>2</td>
</tr>
<tr>
<td>Read Screen Format</td>
<td>3</td>
</tr>
<tr>
<td>Read screen from cursor</td>
<td>4</td>
</tr>
<tr>
<td>WRITE IOCTL</td>
<td></td>
</tr>
<tr>
<td>Write to screen from home</td>
<td>1</td>
</tr>
<tr>
<td>Set Cursor Position</td>
<td>2</td>
</tr>
<tr>
<td>Write to screen from cursor</td>
<td>4</td>
</tr>
</tbody>
</table>
KEYBOARD/VIDEO SUBSYSTEM

SUBSYSTEM: KEYBOARD/VIDEO

Name(fn): Ioctl Read - Read Vidmod

Call
ah = 44H (Ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "VCON" device)
ds:dx = parameter block address

cx = 14 (size of parameter block in bytes)

Parameter Block

<table>
<thead>
<tr>
<th>cmd</th>
<th>dw (?) ; = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>troff</td>
<td>dw (?) ; = address of VIMOD structure</td>
</tr>
<tr>
<td>trcnt</td>
<td>dw (?) ; = 19 (size of VIMOD struc)</td>
</tr>
<tr>
<td>filler</td>
<td>dw 4 dup (?) ;reserved</td>
</tr>
</tbody>
</table>

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
Copies current VIDEO VIMOD structure (see Appendix C) into the address specified by troff in the parameter block.
SUBSYSTEM:        KEYBOARD/VIDEO

Name(fn):        Ioctl Read - Read screen from home

Call
    ah = 44H    (Ioctl fn number)
    al = 2      (function code)
    bx = handle (from OPEN on "VCON" device)
    ds:dx = parameter block address
    cx = 14     (size of parameter block in bytes)

Parameter Block
    struc
        cmd  dw (?) ; = 1
        troff dw (?) ; = address of transfer buffer
        trcnt dw (?) ; = number of bytes to be transferred
        filler dw 4 dup (?) ; reserved
    ends

Return

    Carry set:
    ax = 6  (invalid handle)
            = 1  (invalid function)
            = 13 (invalid data)
            = 5  (access denied)
    Carry not set:
    ax = 14 (size of parameter block in bytes)

Explanation
    Trcnt bytes are read into the transfer buffer from the
    screen starting at the home position.
KEYBOARD/VIDEO SUBSYSTEM

SUBSYSTEM: KEYBOARD/VIDEO

Name(fn): Ioctl Read - Get Cursor Position

Call

\[
\begin{align*}
\text{ah} & = 44H & \text{(ioctl fn number)} \\
\text{al} & = 2 & \text{(function code)} \\
\text{bx} & = \text{handle} & \text{(from OPEN on "VCON" device)} \\
\text{ds:dx} & = \text{parameter block address} \\
\text{cx} & = 14 & \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

```
struc
  cmd       db (?) ; = 2  
  troff     dw (?) ; = address of CPOS structure  
  trcnt     dw (?) ; = size of CPOS structure  
  filler    dw 4 dup (?) ;reserved

ends
```

Return

```
Carry set:
\[
\begin{align*}
\text{ax} & = 6 & \text{(invalid handle)} \\
& = 1 & \text{(invalid function)} \\
& = 13 & \text{(invalid data)} \\
& = 5 & \text{(access denied)}
\end{align*}
\]

Carry not set:
\[
\begin{align*}
\text{ax} & = 14 & \text{(size of parameter block in bytes)}
\end{align*}
\]
```

Explanation

The CPOS structure (see Appendix C) is copied into the buffer at the address specified by 'troff' in the parameter block.
KEYBOARD/VIDEO SUBSYSTEM

SUBSYSTEM: KEYBOARD/VIDEO

Name(fn): Ioctl Read - Read Screen Format

Call

\[
\begin{align*}
ah &= 44H \quad \text{(Ioctl fn number)} \\
\text{al} &= 2 \quad \text{(function code)} \\
\text{bx} &= \text{handle} \quad \text{(from OPEN on "VCON" device)} \\
\text{ds:dx} &= \text{parameter block address} \\
\text{cx} &= 14 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\[
\begin{align*}
\text{struc} & \\
\text{cmd} & \text{dw (?) ; = 3} \\
\text{troff} & \text{dw (?) ; = address of SFRMT struc} \\
\text{trcnt} & \text{dw (?) ; = 4 (size of SFRMT struc)} \\
\text{filler} & \text{dw 4 dup (?) ; reserved}
\end{align*}
\]

\[
\text{ends}
\]

Return

Carry set:

\[
\begin{align*}
\text{ax} &= 6 \quad \text{(invalid handle)} \\
\text{cx} &= 1 \quad \text{(invalid function)} \\
\text{dx} &= 13 \quad \text{(invalid data)} \\
\text{cx} &= 5 \quad \text{(access denied)}
\end{align*}
\]

Carry not set:

\[
\begin{align*}
\text{ax} &= 14 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Explanation

The SFRMT structure (see Appendix C) is copied into the buffer at the address specified by troff in the parameter block.
KEYBOARD/VIDEO SUBSYSTEM

SUBSYSTEM: KEYBOARD/VIDEO

Name(fn): Ioctl Read - Read screen from cursor

Call
ah = 44H (Ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "VCON" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struc
cmd dw (?) ; = 4
troff dw (?) ; = address of transfer buffer
trcnt dw (?) ; = no of bytes to be transferred
filler dw 4 dup (?) ; reserved
endstr

Return

Carry set:
ax = 6 (invalid handle)
     = 1 (invalid function)
     = 13 (invalid data)
     = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block)

Explanation
Trcnt bytes are copied into the transfer buffer from the screen starting at the current cursor position.
KEYBOARD/VIDEO SUBSYSTEM

SUBSYSTEM: KEYBOARD/VIDEO

Name(fn): Ioctl Write - Write to screen from home

Call
ah = 44H  (Ioctl fn number)
al = 3    (function code)
bx = handle (from OPEN on "VCON" device)
ds:dx = parameter block address

cx = 14    (size of parameter block in bytes)

Parameter Block
struc
cmd    dw (?) ; = 1
troff  dw (?) ; = address of transfer buffer
trcnt  dw (?) ; = no. of bytes to be transferred
filler dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
     = 1 (invalid function)
     = 13 (invalid data)
     = 5 (access denied)
Carry not set:
ax = 14    (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the screen starting at the home position.
KEYBOARD/VIDEO SUBSYSTEM

SUBSYSTEM: KEYBOARD/VIDEO

Name(fn): Ioctl Write - Set Cursor Position

Call

\[
\begin{align*}
\text{ah} &= 44H \quad \text{(Ioctl fn number)} \\
\text{al} &= 3 \quad \text{(function code)} \\
\text{bx} &= \text{handle} \quad \text{(from OPEN on " VCON" device)} \\
\text{ds:dx} &= \text{parameter block address} \\
\text{cx} &= 14 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\[
\text{struc}
\begin{align*}
\text{cmd} &\quad \text{dw (?) ; } = 2 \\
\text{troff} &\quad \text{dw (?) ; } = \text{address of CPOS structure} \\
\text{trcnt} &\quad \text{dw (?) ; } = \text{size of CPOS structure} \\
\text{filler} &\quad \text{dw 4 dup (?) ; reserved}
\end{align*}
\text{ends}
\]

Return

Carry set:

\[
\begin{align*}
\text{ax} &= 6 \quad \text{(invalid handle)} \\
\text{ax} &= 1 \quad \text{(invalid function)} \\
\text{ax} &= 13 \quad \text{(invalid data)} \\
\text{ax} &= 5 \quad \text{(access denied)}
\end{align*}
\]

Carry not set:

\[
\begin{align*}
\text{ax} &= 14 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Explanation

Sets the cursor position to the value set up in the CPOS structure (see Appendix C).

Note. This IOCTL call will not be put into effect if the ANSI field of the VIMOD structure is active.
KEYBOARD/VIDEO SUBSYSTEM

SUBSYSTEM: KEYBOARD/VIDEO

Name(fn): Ioctl Write - Write to screen from cursor

Call

ah = 44H (Ioctl fn number)
al = 3 (function code)
bx = handle (from OPEN on "VCON" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block

struc
cmd    dw (?) ; = 4
troff  dw (?) ; = address of transfer buffer
trcnt  dw (?) ; = no of bytes to be transferred
filler dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
    = 1 (invalid function)
    = 13 (invalid data)
    = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the screen starting at the current cursor position.
## Supported ioctl Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ IOCTL</td>
<td></td>
</tr>
<tr>
<td>Read Vidmod</td>
<td>0</td>
</tr>
<tr>
<td>Read screen from home</td>
<td>1</td>
</tr>
<tr>
<td>Get Cursor Position</td>
<td>2</td>
</tr>
<tr>
<td>Read Screen Format</td>
<td>3</td>
</tr>
<tr>
<td>Read screen from cursor</td>
<td>4</td>
</tr>
</tbody>
</table>

| WRITE IOCTL                 |      |
| Write to screen from home   | 1    |
| Set Cursor Position         | 2    |
| Write to screen from cursor | 4    |
SUBSYSTEM: VIDEO

Name(fn): ioctl Read - Read screen from home

Call

\[
\begin{align*}
\text{ah} & = 44H \quad \text{(ioctl fn number)} \\
\text{al} & = 2 \quad \text{(function code)} \\
\text{bx} & = \text{handle} \quad \text{(from OPEN on "VID" device)} \\
\text{ds:dx} & = \text{parameter block address} \\
\text{cx} & = 14 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\[
\begin{align*}
\text{struc} \\
\quad \text{cmd} & \quad \text{dw (?) ; } = 1 \\
\quad \text{troff} & \quad \text{dw (?) ; } = \text{address of transfer buffer} \\
\quad \text{trcnt} & \quad \text{dw (?) ; } = \text{no of bytes to be transferred} \\
\quad \text{filler} & \quad \text{dw 4 dup (?) ; reserved}
\end{align*}
\]

\text{ends}

Return

\[
\begin{align*}
\text{Carry set:} \\
\quad \text{ax} & = 6 \quad \text{(invalid handle)} \\
& = 1 \quad \text{(invalid function)} \\
& = 13 \quad \text{(invalid data)} \\
& = 5 \quad \text{(access denied)}
\end{align*}
\]

\[
\begin{align*}
\text{Carry not set:} \\
\quad \text{ax} & = 14 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Explanation

Trcnt bytes are read into the transfer buffer from the screen starting at the home position.
VIDEO SUBSYSTEM

SUBSYSTEM: VIDEO

Name(fn): Ioctl Read - Get Cursor Position

Call

\[
\begin{align*}
\text{ah} &= 44H \quad \text{(Ioctl fn number)} \\
\text{al} &= 2 \quad \text{(function code)} \\
\text{bx} &= \text{handle} \quad \text{(from OPEN on "VID" device)} \\
\text{ds:dx} &= \text{parameter block address} \\
\text{cx} &= 14 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\[
\text{struc} \\
\text{cmd} & \quad \text{db (?) ; = 2} \\
\text{troff} & \quad \text{dw (?) ; = address of CPOS structure} \\
\text{trcnt} & \quad \text{dw (?) ; = size of CPOS structure} \\
\text{filler} & \quad \text{dw 4 dup (?) ; reserved} \\
\text{ends}
\]

Return

\[
\begin{align*}
\text{Carry set:} \\
\text{ax} &= 6 \quad \text{(invalid handle)} \\
&= 1 \quad \text{(invalid function)} \\
&= 13 \quad \text{(invalid data)} \\
&= 5 \quad \text{(access denied)}
\end{align*}
\]

Carry not set:

\[
\text{ax} = 14 \quad \text{(size of parameter block in bytes)}
\]

Explanation

The CPOS structure (see Appendix C) is read into the buffer at the address specified by 'troff' in the parameter block.
VIDEO SUBSYSTEM

SUBSYSTEM: VIDEO

Name(fn): Ioctl Read - Read Screen Format

Call
ah = 44H (Ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "VID" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struc
cmd
dw (?); = 3
troff
dw (?); = address of SFRMT struc
trcnt
dw (?); = 4 (size of SFRMT struc)
filler
dw 4 dup (?); reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
   = 1 (invalid function)
   = 13 (invalid data)
   = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
The SFRMT structure (see Appendix C) is copied into the buffer at the address specified by troff in the parameter block.
VIDEO SUBSYSTEM

SUBSYSTEM: VIDEO

Name(fn): Ioctl Read - Read screen from cursor

Call
ah = 44H (ioctl fn number)
al = 2 (function code)
bx = handle (from OPEN on "VID" device)
ds:dx = parameter block address
cx = 14 (size of parameter block in bytes)

Parameter Block
struc
cmd
dw (?) ; = 4
troff
dw (?) ; = address of transfer buffer
trcnt
dw (?) ; = no of bytes to be transffered
filler
dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
     = 1 (invalid function)
     = 13 (invalid data)
     = 5 (access denied)

Carry not set:
ax = 14 (size of parameter block in bytes)

Explanation
Trcnt bytes are copied into the transfer buffer from the screen starting at the current cursor position.
VIDEO SUBSYSTEM

SUBSYSTEM: VIDEO

Name(fn): Ioctl Write - Write to screen from home

Call

ah = 44H  (Ioctl fn number)
cl = 3    (function code)
bx = handle  (from OPEN on "VID" device)
ds:dx = parameter block address
cx = 14  (size of parameter block in bytes)

Parameter Block
struc
cmd         dw (?) ; = 1
troff       dw (?) ; = address of transfer buffer
trcnt       dw (?) ; = no. of bytes to be transferred
filler      dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
  = 1  (invalid function)
  = 13 (invalid data)
  = 5  (access denied)
Carry not set:
ax = 14  (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the screen starting at the home position.
VIDEO SUBSYSTEM

SUBSYSTEM: VIDEO

Name(fn): Ioctl Write - Set Cursor Position

Call

ah = 44H  (Ioctl fn number)
al = 3     (function code)
bx = handle (from OPEN on "VID" device)
ds:dx = parameter block address
cx = 14    (size of parameter block in bytes)

Parameter Block

struc
    cmd      dw (?) ; = 2
    troff    dw (?) ; = address of CPOS structure
    trcnt    dw (?) ; = size of CPOS structure
    filler   dw 4 dup (?) ; reserved
ends

Return

Carry set:
ax = 6 (invalid handle)
     = 1 (invalid function)
     = 13 (invalid data)
     = 5 (access denied)

Carry not set:
ax = 14    (size of parameter block in bytes)

Explanation

Sets the cursor position to the value set up in the CPOS structure (see Appendix C).

Note. This IOCTL function will not update the CPOS structure if the ANSI flag is set in the VIMOD structure.
VIDEO SUBSYSTEM

SUBSYSTEM: VIDEO

Name(fn): IOCTL Write - Write to screen from cursor

Call

ah = 44H  (IOctl fn number)
al = 3    (function code)
bx = handle  (from OPEN on "VID" device)
ds:dx = parameter block address

Size = 14  (size of parameter block in bytes)

Parameter Block
struct
    cmd    dw (?);   = 4
    troff   dw (?); = address of transfer buffer
    trcnt  dw (?);  = no of bytes to be transferred
        filler dw 4 dup (?); reserved
ends

Return

Carry set:
ax = 6  (invalid handle)
    = 1  (invalid function)
    = 13 (invalid data)
    = 5  (access denied)

Carry not set:
ax = 14  (size of parameter block in bytes)

Explanation
Trcnt bytes from the transfer buffer are written to the screen starting at the current cursor position.
<table>
<thead>
<tr>
<th>SECTION</th>
<th>FUNCTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Aux</td>
<td>Read Uart Configuration</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Set Uart Configuration</td>
<td>4</td>
</tr>
<tr>
<td>Aux2</td>
<td>Read Uart Configuration</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Set Uart Configuration</td>
<td>7</td>
</tr>
<tr>
<td>Clock</td>
<td>Get Alarm</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Set Time</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Get Time</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Set Alarm</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Delete Alarm</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Wait for Alarm/Interrupt</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Time of Next Alarm</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Scan Alarms</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Read Clock Configuration</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Floppy Disk</td>
<td>Read/Write disk track</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Read Floppy Disk Configuration</td>
<td>20</td>
</tr>
<tr>
<td>Keyboard</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Read Keyboard Configuration</td>
<td>22</td>
</tr>
<tr>
<td>Lcd</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Read Lcd Configuration</td>
<td>24</td>
</tr>
<tr>
<td>Ram</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Format Ram Disk</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Read Ram Driver Configuration</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Return Max Ram Disk Size</td>
<td>28</td>
</tr>
<tr>
<td>Rom-Ram</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Return Device Base Address</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Read Rom-Ram Configuration</td>
<td>31</td>
</tr>
<tr>
<td>Video</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Read Video Configuration</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>SECTION</td>
<td>FUNCTION</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Operating System</td>
<td>Read OS configuration</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>Read Off Key enable state</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Write Off key enable state</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>Read Video enable state</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Read Shutdown enable state</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Write Shutdown enable state</td>
<td>41</td>
</tr>
<tr>
<td>Printer Driver</td>
<td>Read PR Configuration</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>44</td>
</tr>
</tbody>
</table>
INTRODUCTION

A Bicall is a mechanism by which direct BIOS functions can be performed from application programs. Bicalls use the concept of a two level vectoring system based on the driver SUBSYSTEM and the driver FUNCTION. For each Subsystem there are 12 standard functions and some non standard functions. Documentation for the standard MSDOS driver functions can be found in the MSDOS Operating System Programmer’s Reference manual, chapter 2 section 2.6 on device drivers.

Table 1 contains a summary of the non standard functions. This chapter documents these NON standard driver functions supported by the DULMONT MAGNUM for each Subsystem.

A Bicall uses interrupt INT 254 to interface with the BIOS. Parameters are passed to and from the driver function via registers and a parameter block set up before the interrupt is called. The call conventions are

\begin{align*}
es:bx &= \text{address of parameter block} \\
ax &= \text{16 bit Subsystem Code} \\
ni &= \text{16 bit Function Code} \\
di &= \text{size of parameter block}
\end{align*}

On return if bx contains the value 0 then the results of the driver function are returned by either updating the parameter block and/or are stored in the register pair ax/dx. Otherwise if bx contains the value 1 or 2 invalid parameters were passed to the bicall and no function will have been performed. If an invalid subsystem code was used bx will contain the error code – 1. If a non allocated function code was used (i.e. one within the range of the driver function table but with a null function address) bx will contain the error code – 2. N.B. If a function code that is out of range is used the system will crash.

For example a bicall to the Clock Driver’s Subsystem to invoke the Get Time function must have the equates and the parameter block structure as shown in the following sample program. On return bx holds the status returned by the driver and the parameter block will hold the current time if no errors occurred.
INTRODUCTION

SAMPLE PROGRAM - Using Clock Driver Bicall

IBIOS NUM equ 254 ; bicall interrupt
PBLKSIZE equ 6 ; size of parameter block
SSYS CC equ 0 ; clock driver subsystem
GET TIM equ 15 ; code for function
NOERR equ 0 ; no error return status

PBLK struc
day dw (?) ; days since 1/1/1980
min db (?) ; minutes
hour db (?) ; hours
hsec db (?) ; hundredths of sec
sec db (?) ; seconds

PBLK ends

mov ax,ds
mov es,ax
mov bx,PBLK
mov di,PBLKSIZE
mov ax,SSYS CC
mov si,GET TIM
int IBIOS NUM
cmp bx, NOERR
jne ERR
;
;
ERR ret

; set up registers
; call interrupt
; test for error
; can use PBLK values to
; print time for example
AUX SUBSYSTEM

SUBSYSTEM: AUX DRIVER

FUNCTION: Read AUX Configuration

Call

\[
\begin{align*}
ax &= 6 \quad \text{(subsystem number)} \\
sl &= 13 \quad \text{(function code)} \\
es:bx &= \text{parameter block address} \\
di &= 8 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\[
\text{struct}
\]

\[
\begin{align*}
\text{version} &\quad \text{dw (?) ;version number of the AUX driver} \\
\text{baud0} &\quad \text{db (?) ;baud rate for uart0} \\
\text{par0} &\quad \text{db (?) ;parity for uart0} \\
\text{data0} &\quad \text{db (?) ;data bits for uart0} \\
\text{baud1} &\quad \text{db (?) ;baud rate for uart1} \\
\text{par1} &\quad \text{db (?) ;parity for uart1} \\
\text{data1} &\quad \text{db (?) ;data bits for uart1}
\end{align*}
\]

\[
\text{ends}
\]

Return

\[
\begin{align*}
bx &= 1 \text{ if subsystem invalid} \\
&= 2 \text{ if invalid function} \\
&= 0 \text{ if successful bcall}
\end{align*}
\]

On successful bcall

Parameter Block = current AUX configuration.

Explanation

Reads the current UART configuration for both uart subsystems into the parameter block.
(See Appendix D for baud rate, parity and data bits codes.) The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level.
AUX SUBSYSTEM

SUBSYSTEM: AUX DRIVER
FUNCTION: Set AUX Configuration

Call
ax = 6  (subsystem number)
si = 14 (function code)
es:bx = parameter block address
di = 8  (size of parameter block in bytes)

Parameter
struc
  version      dw (?) ;version number of the AUX driver
  baud0       db (?) ;(0-7) baud rate for uart0
  par0        db (?) ;(0-2) parity for uart0
  data0       db (?) ;(0,4,8,12,16,20,24,28)
              ;data bits for uart0
  baud1       db (?) ;(0-7) baud rate for uart1
  par1        db (?) ;(0-2) parity for uart1
  data1       db (?) ;(0,4,8,12,16,20,24,28)
              ;data bits for uart1
ends

Return
Nothing

Explanation

Set the Uart for both uart subsystems to the configuration in the parameter block. (See the Appendix D for baudrate, parity and databit codes.) No checking is done on the data in the parameter block.
AUX2 SUBSYSTEM

This subsystem handles calls to the Uart driver for the AUX2 port.

Supported Function Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uart Configuration</td>
<td>12</td>
</tr>
<tr>
<td>Set Uart Configuration</td>
<td>14</td>
</tr>
</tbody>
</table>
CLOCK SUBSYSTEM

SUBSYSTEM: Clock Driver
FUNCTION: Get Alarm

Call

\[
\begin{align*}
ax &= 0 \quad \text{(subsystem number)} \\
\text{si} &= 13 \quad \text{(function code)} \\
\text{es:bx} &= \text{parameter block address} \\
dl &= 2 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\[
\begin{align*}
\text{struc} \\
\quad \text{alarm handle} & \quad \text{dw (?)} \quad ; = \text{alarm handle} \\
\text{ends}
\end{align*}
\]

Return

\[
\begin{align*}
\text{bx} &= 1 \text{ if subsystem invalid} \\
&= 2 \text{ if invalid function} \\
&= 0 \text{ if successfull bicall}
\end{align*}
\]

On successfull bicall

\[
\begin{align*}
\text{dx:ax} &= \text{NULL (0) on failure of function} \\
&= \text{segment/offset address of ALARM structure on success}
\end{align*}
\]

Explanation

Returns the segment/offset address of the ALARM structure for the alarm identified by the alarm handle.

On entry the alarm handle field in the parameter block must be a handle returned from a previous Set Alarm or Next Alarm bicall.

On return the register pair dx:ax is NULL if the specified alarm handle is non existent. Otherwise dx:ax holds the address of the specified alarm's structure.
CLOCK SUBSYSTEM

SUBSYSTEM: Clock Driver

FUNCTION: Set Time

Call

ax = 0 (subsystem number)
sl = 14 (function code)
es:bx = parameter block address
di = 6 (size of parameter block in bytes)

Parameter Block

struct
day dw (?) ; = days since 1/1/80
min db (?) ; = 0-59 (minutes)
hour db (?) ; = 0-23 (hours)
hsec db (?) ; = 0-99 (hundredths of sec)
sec db (?) ; = 0-59 (seconds)
ends

Return
bx = 1 if subsystem invalid
     2 if invalid function
     0 if successfull bicall
On successfull bicall
ax = 1 on success
    NULL (0) on failure

Explanation

Sets the current time from the values in the parameter block.

On entry the parameter block holds the time to be set.

On exit ax is equal to 1 on success and equal to NULL if the parameter block held invalid time values.
CLOCK SUBSYSTEM

SUBSYSTEM: Clock Driver

FUNCTION: Get Time

Call
ax = 0  (subsystem number)
si = 15  (function code)
es:bx = parameter block address
di = 6   (size of parameter block in bytes)

Parameter Block
struc
day   dw (?) ;days since 1/1/80
min   db (?) ;minutes
hour  db (?) ;hours
hsec  db (?) ;hundredths of sec
sec   db (?) ;seconds
ends

Return
bx = 1 if subsystem invalid  
   = 2 if invalid function  
   = 0 if successfully bcall
On successful bcall
Parameter Block = filled in

Explanation

Reads the current time.

On return the Parameter Block is filled in to indicate the current time.
SUBSYSTEM: Clock Driver

FUNCTION: Set Alarm

Call
ax = 0 (subsystem number)
si = 16 (function code)
es:bx = parameter block address
di = 14 (size of parameter block in bytes)

Parameter Block
struct
  alarm dw size ALARM dup (?) ; = ALARM struct
ends

Return
bx = 1 if subsystem invalid
     = 2 if invalid function
     = 0 if successful bcall
On successful bcall
ax = alarm handle on success
     = NULL (0) on failure

Explanation

Returns the alarm handle for the newly set alarm.

On entry the alarm field in the parameter block is an Alarm structure holding all the appropriate alarm data (see Appendix A).

On return ax is NULL if an invalid time was held in the ALARM structure 'alarm', the maximum number of alarms had already been set (see Appendix A), or an attempt was made to set a noisy, repetitive alarm. Otherwise ax is the handle by which the newly set alarm can be identified.
CLOCK SUBSYSTEM

SUBSYSTEM: Clock Driver

FUNCTION: Delete Alarm

Call

\[
\begin{align*}
ax &= 0 \quad \text{(subsystem number)} \\
sl &= 17 \quad \text{(function code)} \\
\text{es:bx} &= \text{parameter block address} \\
\text{dl} &= 2 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block
struc

\[
\begin{align*}
\text{alarm handle} & \quad \text{dw (?) ; = alarm handle}
\end{align*}
\]
ends

Return

\[
\begin{align*}
\text{bx} &= 1 \text{ if subsystem invalid} \\
&= 2 \text{ if invalid function} \\
&= 0 \text{ if successful bcall}
\end{align*}
\]

On successful bcall
\[
\begin{align*}
\text{ax} &= \text{alarm handle on success} \\
&= \text{NULL (0) on failure}
\end{align*}
\]

Explanation

Deletes the alarm identified by 'alarm handle' in the parameter block.

On entry the alarm handle field in the parameter block holds the handle returned by a Set Alarm or Next bcall.

On return ax holds the status. If no alarm existed ax holds NULL otherwise ax holds the 'alarm handle' just deleted.
CLOCK SUBSYSTEM

SUBSYSTEM: Clock Driver

FUNCTION: Wait for interrupt

Call

\[
\begin{align*}
ax &= 0 \quad \text{(subsystem number)} \\
si &= 18 \quad \text{(function code)} \\
di &= \varnothing \\
es &= ds
\end{align*}
\]

Return

\[
\begin{align*}
bx &= 1 \text{ if subsystem invalid} \\
&= 2 \text{ if invalid function} \\
&= 0 \text{ if successful bcall}
\end{align*}
\]

Explanation

Returns when any interrupt occurs. In the interim it powers down if possible.
CLOCK SUBSYSTEM

SUBSYSTEM: Clock Driver

FUNCTION: Next Alarm Handle

Call
ax = 0 (subsystem number)
si = 19 (function code)
es:bx = parameter block address
di = 2 (size of parameter block in bytes)

Parameter Block
struc
    alarm handle  dw (?) ; = alarm handle
ends

Return
bx = 1 if subsystem invalid
     = 2 if invalid function
     = 0 if successful bcall

On successful bcall
ax = alarm handle on success
    = NULL on failure

Explanation

Returns the alarm handle for the next scheduled alarm after the alarm identified by the 'alarm handle' in the parameter block.

If the 'alarm handle' is NULL then the alarm handle for the first scheduled alarm is returned.

On return ax is NULL if either the alarm handle is invalid or there is no next scheduled alarm. Otherwise ax is the alarm handle by which the next scheduled alarm can be identified.
CLOCK SUBSYSTEM

SUBSYSTEM: Clock Driver

FUNCTION: Read Clock Configuration

Call
\[
\begin{align*}
ax &= 0 \quad \text{(subsystem number)} \\
si &= 21 \quad \text{(function code)} \\
\text{es:bx} &= \text{parameter block address} \\
di &= 2 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block
\[
\begin{align*}
\text{version} &\quad \text{dw (?) ;version number}
\end{align*}
\]

Return
\[
\begin{align*}
bx &= 1 \text{ if subsystem invalid} \\
&= 2 \text{ if invalid function} \\
&= 0 \text{ if successful bcall}
\end{align*}
\]

On successful bcall
\[
\begin{align*}
\text{version} &= \text{current clock driver version number}
\end{align*}
\]

Explanation

Read the clock driver's configuration.

On exit 'version' in the parameter block contains the version number for the clock driver as a 2 byte code. The high order byte is the version number and the low order byte is the edit level.
**FLOPPY DISK SUBSYSTEM**

**Supported Function Calls**

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Disk Track</td>
<td>13</td>
</tr>
<tr>
<td>Read FD Configuration</td>
<td>14</td>
</tr>
</tbody>
</table>
Floppy Disk Subsystem

Subsystem: Floppy Disk

Function: Write Disk Track

Call

\[ \begin{align*}
  ax & = 1 \quad \text{(subsystem number)} \\
  si & = 13 \quad \text{(function code)} \\
  es:bx & = \text{parameter block address} \\
  di & = 18 \quad \text{(size of parameter block in bytes)}
\end{align*} \]

Parameter Block (MSDOS read/write parameter block)

\[ \begin{align*}
  \text{struc} & \\
  \text{filler} & \text{db 3dup(?) reserved} \\
  \text{status} & \text{dw (?) status of call} \\
  \text{filler} & \text{db 5dup(?) reserved} \\
  \text{formatlp} & \text{dd (?) offset/segment pair for FORMAT struc} \\
  \text{filler} & \text{db 4dup(?) reserved}
\end{align*} \]

\[ \text{ends} \]

FORMAT struc

\[ \begin{align*}
  \text{cmd} & \text{db (?) } = 0 \text{(wttrack)} \\
  \text{track} & \text{dw (?) } = 0-39 \text{ (track number)} \\
  \text{sideno} & \text{dw (?) } = 0-1 \text{ (side no.)} \\
  \text{datalp} & \text{dd (?) offset/segment pair for transfer buffer} \\
  \text{unit} & \text{db (?) } = 0,1,2.. \text{ (unit no.)} \\
  \text{trcnt} & \text{dw (?) transfer count} \\
  \text{status} & \text{dw (?) status of call (see App B)}
\end{align*} \]

FORMAT end

Return

\[ \begin{align*}
  bx & = 1 \text{ if subsystem invalid} \\
  & = 2 \text{ if invalid function} \\
  & = 0 \text{ if successful bical} \\
\end{align*} \]

On successful bical

\[ \begin{align*}
  \text{status} & = \text{return status (see App B)} \\
  \text{transfer buffer} & = \text{contains bytes read when READ issued} \\
  \text{trcnt} & = \text{no of bytes not transferred}
\end{align*} \]

Explanation

Writes a disk track from the transfer buffer, providing an interface to FORMAT programs for the WERTERN DIGITAL chip.
On entry 'formatlp' in the parameter block addresses a
FORMAT structure which is set up as follows. 'Cmd'
specifies that a write is to be performed. 'Track',
'sideno' and 'unit' state from where the transfer is to
be commenced. 'Sideno' is set as 0 to specify side one
and 1 to specify side two. 'Unit' is set to the floppy
disk unit you wish to access (i.e. 0 for drive e:, 1 for
drive f:, etc.). 'Datalp' is the address of the transfer
buffer. 'Trcnt' must be greater than the maximum number
of bytes written to a track during a track write
operation as specified in the Western Digital's Hardware
Reference. The transfer buffer must contain the data to
format the track set up to IBM 3740 standard.
On return the 'status' in the parameter block and in the
FORMAT structure is identical showing whether a write or
disk error has occurred. The 'trcnt' value shows the
number of bytes not written.
SUBSYSTEM: Floppy Disk

FUNCTION: Read FD Configuration

Call

\[ ax = 1 \]
\[ si = 14 \]
\[ es:bx = \text{parameter block address} \]
\[ di = 16 \quad \text{(size of parameter block in bytes)} \]

Parameter Block

```
struc
  version  dw  ;version number
  filler1  dw  ;reserved
  nunits   dw  ;no of drive units
  moff     dw  ;motor off delay time (secs)
  filler2  dd  ;4 bytes reserved
  pbpblp   dd  ;segment-offset address to
              ;FDBPB ptr table
endstruc
```

Return

\[ bx = 1 \text{ if subsystem invalid} \]
\[ = 2 \text{ if invalid function} \]
\[ = 0 \text{ if successful bcall} \]

On successful bcall
\[ \text{parameter block} = \text{Current configuration}. \]

Explanatıon

Fills in the parameter block.

On entry the parameter block is empty.

The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level. The 'pbpblp' is the address of a table of addresses to FDBPB structures (see Appendix B) where there exists one entry for each floppy disk unit, indexed by unit number (0,1,2,etc).
### Supported Function Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Keyboard Configuration</td>
<td>13</td>
</tr>
</tbody>
</table>
KEYBOARD SUBSYSTEM

SUBSYSTEM: Keyboard Driver

FUNCTION: Read Keyboard Driver Configuration

Call
ax = 2 (subsystem number)
si = 13 (function code)
es:bx = parameter block address
di = 2 (size of parameter block in bytes)

Parameter Block
struc
    version
    dw (?) ;version number
ends

Return
bx = 1 if subsystem invalid
    = 2 if invalid function
    = 0 if successful bicalc

On successfull bicalc
version = current keyboard driver version number

Explanation

Read the keyboard driver's configuration.

The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level.
## LCD SUBSYSTEM

### Supported Function Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read LCD Configuration</td>
<td>13</td>
</tr>
</tbody>
</table>
SUBSYSTEM: LCD

FUNCTION: Read LCD Driver Configuration

Call

\[ \begin{align*}
    ax &= 4 \quad \text{(subsystem number)} \\
    si &= 13 \quad \text{(function code)} \\
    es:bx &= \text{parameter block address} \\
    di &= 8 \quad \text{(size of parameter block in bytes)}
\end{align*} \]

Parameter Block

\begin{verbatim}
struc
version     dw(?) ; version number
smode       db(?) ; screen mode
wrap        db(?) ; wrap mode
vimodlp     dd(?) ; segment-offset to VIMOD struc
ends
\end{verbatim}

Return

\[ \begin{align*}
    bx &= 1 \text{ if subsystem invalid} \\
    &= 2 \text{ if invalid function} \\
    &= 0 \text{ if successful bcall}
\end{align*} \]

On successful bcall

parameter block = Current configuration.

Explanation

Updates the configuration structure.

The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level. For screen mode and wrap mode definitions see Appendix C - LCSCRN structure.

Vimodlp' addresses a VIMOD structure (see Appendix C) holding all the current screen information.
### RAM SUBSYSTEM

#### Supported Function Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format Ram Disk</td>
<td>13</td>
</tr>
<tr>
<td>Read Ram Driver Configuration</td>
<td>14</td>
</tr>
<tr>
<td>Return Max Ram Disk Size</td>
<td>15</td>
</tr>
</tbody>
</table>
RAM SUBSYSTEM

SUBSYSTEM: Ram Driver

FUNCTION: Format Ram Disk to Maximum Size

Call
ax = 11 (subsystem number)
si = 13 (function code)
bx = 0
di = 0

Return
bx = 1 if subsystem invalid
     2 if invalid function
     0 if successful bcall

Explanation
Formats the ram disk to the maximum size possible depending on the amount of ram in the system configuration. (See RAM driver manual).
RAM SUBSYSTEM

SUBSYSTEM: Ram Driver

FUNCTION: Read Ram Driver Configuration

Call

\[
\begin{align*}
ax &= 11 \quad \text{(subsystem number)} \\
si &= 14 \quad \text{(function code)} \\
es:bx &= \text{parameter block address} \\
di &= 2 \quad \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\[
\begin{align*}
\text{struc} \\
\text{version} \\
\text{ends}
\end{align*}
\]

Return

\[
\begin{align*}
bx &= 1 \text{ if subsystem invalid} \\
&= 2 \text{ if invalid function} \\
&= 0 \text{ if successful bcall}
\end{align*}
\]

On successful bcall

\[
\text{version} = \text{current Ram driver version number}
\]

Explanation

Read the Ram driver's configuration.

The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level.
**RAM SUBSYSTEM**

**SUBSYSTEM:** Ram Drive

**FUNCTION:** Get Ram disk size

**Call**

\[
\begin{align*}
ax &= 11 & \text{(subsystem number)} \\
si &= 15 & \text{(function code)} \\
bx &= 0 \\
di &= 0
\end{align*}
\]

**Return**

\[
\begin{align*}
bx &= 1 \text{ if subsystem invalid} \\
     &= 2 \text{ if invalid function} \\
     &= 0 \text{ if successful bcall} \\
\text{On successful bcall} \\
ax &= \text{maximum size of Ram Disk}
\end{align*}
\]

**Explanation**

Reads the size for the Ram disk dependent on the amount of ram in the system configuration.

On exit ax holds the maximum size of ram disk.
## Supported Function Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Device Base Address</td>
<td>13</td>
</tr>
<tr>
<td>Read Rom-Ram Driver Configuration</td>
<td>14</td>
</tr>
</tbody>
</table>
ROM/RAM SUBSYSTEM

SUBSYSTEM: Rom-ram Driver

FUNCTION: Return device base address

Call

\[ \begin{align*}
ax & = 3 \quad \text{(subsystem number)} \\
si & = 13 \quad \text{(function code)} \\
es:bx & = \text{parameter block address} \\
di & = 2
\end{align*} \]

Parameter Block

\[
\text{struc}
\begin{align*}
\text{devno} & \quad \text{dw(?); device number (0-2)}
\end{align*}
\text{ends}
\]

Return

\[
\begin{align*}
\text{bx} & = 1 \text{ if subsystem invalid} \\
& = 2 \text{ if invalid function} \\
& = 0 \text{ if successful bcall}
\end{align*}
\]

On successful bcall

\[
\text{dx:ax} = \text{segment-offset address of Base of specified Rom-ram device}
\]

Explanation

Returns base address for Rom-ram device specified in the parameter block.

On entry 'devno' is set to a Rom-ram device number. The valid device numbers are ROM0 - 0, ROM1 - 1 and ROM2 - 2.

On exit dx:ax holds the segment-offset base address for the specified device.
POM/RAM subsystem

Subsystem: Rom-ram Driver

Function: Read Rom-ram Driver Configuration

Cell

\[
\begin{align*}
ax &= 5 & \text{(subsystem number)} \\
si &= 14 & \text{(function code)} \\
es:bx &= \text{parameter block address} \\
di &= 2 & \text{(size of parameter block in bytes)}
\end{align*}
\]

Parameter Block

\begin{verbatim}
struc
  version
dw (?); version number

ends
\end{verbatim}

Return

\[
\begin{align*}
& bx = 1 \text{ if subsystem invalid} \\
& \quad = 2 \text{ if invalid function} \\
& \quad = 0 \text{ if successful bcall}
\end{align*}
\]

On successful bcall

version = current Rom-Ram driver version number

Explanation

Read the Rom-ram driver's configuration.

The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level.
**Supported Function Calls**

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Video Driver Configuration 13</td>
<td></td>
</tr>
</tbody>
</table>
VIDEO SUBSYSTEM

SUBSYSTEM:       Video

FUNCTION:       Read Video Driver Configuration

Call

\[
\begin{align*}
\text{ax} &= 8 \quad \text{(subsystem number)} \\
\text{si} &= 13 \quad \text{(function code)} \\
\text{es:bx} &= \text{parameter block address} \\
\text{di} &= 6 \quad \text{(size of parameter block in bytes)} \\
\end{align*}
\]

Parameter Block

\[
\begin{align*}
\text{struct} & \\
\text{version} & \text{dw(?)} \quad \text{; version number} \\
\text{vimodlp} & \text{dd(?)} \quad \text{; segment-offset to VIMOD struc} \\
\text{end} & \\
\end{align*}
\]

Return

\[
\begin{align*}
\text{bx} &= 1 \text{ if subsystem invalid} \\
&= 2 \text{ if invalid function} \\
&= 0 \text{ if successful bcall} \\
\end{align*}
\]

On successful bcall
parameter block = Current configuration.

Explanation

Updates the configuration structure.

The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level.

'Vimodlp' contains the segment-offset address of the Video's VIMOD structure (see Appendix C).
Supported Function Calls

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read OS Configuration</td>
<td>3</td>
</tr>
<tr>
<td>Read Off Key Enable State</td>
<td>4</td>
</tr>
<tr>
<td>Write Off Key Enable State</td>
<td>5</td>
</tr>
<tr>
<td>Read Video Enable State</td>
<td>6</td>
</tr>
<tr>
<td>Read Shutdown Enable State</td>
<td>7</td>
</tr>
<tr>
<td>Write Shutdown Enable State</td>
<td>8</td>
</tr>
</tbody>
</table>
OPERATING SYSTEM SUBSYSTEM

SUBSYSTEM: Operating System
FUNCTION: Read OS Configuration

Call

ax = 9 (subsystem number)
si = 3 (function code)
es:bx = parameter block address
di = 2 (size of parameter block in bytes)

Parameter Block
struc
version db(?); version number
end

Return

bx = 1 if subsystem invalid
     = 2 if invalid function
     = 0 if successful bcall
On successful bcall
parameter block = filled in

Explanation

Reads the operating system version number into the parameter block.

The version number is a 2 byte code. The high order byte is the version number and the low order byte is the edit level.
OPERATING SYSTEM SUBSYSTEM

SUBSYSTEM: Operating System
FUNCTION: Read Off Key Enable State

Call

\[ \begin{align*}
  ax & = 9 & \text{(subsystem number)} \\
  si & = 4 & \text{(function code)} \\
  es:bx & = \text{parameter block address} \\
  di & = 2 & \text{(size of parameter block in bytes)}
\end{align*} \]

Parameter Block

\begin{verbatim}
struct
  state dw(?);
end
\end{verbatim}

Return

\[ \begin{align*}
  bx & = 1 \text{ if subsystem invalid} \\
       & = 2 \text{ if invalid function} \\
       & = 0 \text{ if successful bcall}
\end{align*} \]

On successful bcall

\[ \begin{align*}
  state & = 0 \text{ if Off key disabled} \\
         & = 1 \text{ if Off key enabled}
\end{align*} \]

Explanation

Check if the Off/Reset key is enabled or not and set 'state' appropriately.
OPERATING SYSTEM SUBSYSTEM

SUBSYSTEM: Operating System
FUNCTION: Write Off Key Enable State

Call

ax = 9     (subsystem number)
si = 5     (function code)
es:bx = parameter block address
di = 2     (size of parameter block in bytes)

Parameter Block

struct
  state    dw(?);  (0-disabled, 1-enabled)
end

Return

bx = 1 if subsystem invalid
    = 2 if invalid function
    = 0 if successful bcall

Explanation

Enable or disable the Off/Reset key as specified in the parameter block.
OPERATING SYSTEM SUBSYSTEM

SUBSYSTEM: Operating System
FUNCTION: Read Video Enable State

Call

\[ \begin{align*}
\text{ax} & = 9 \quad \text{(subsystem number)} \\
\text{si} & = 6 \quad \text{(function code)} \\
\text{es:bx} & = \text{parameter block address} \\
\text{di} & = 2 \quad \text{(size of parameter block in bytes)}
\end{align*} \]

Parameter Block

\text{struct}

\begin{align*}
\text{state} & \text{ dw(?) ;} \\
\text{end}
\end{align*}

Return

\[ \begin{align*}
\text{bx} & = 1 \text{ if subsystem invalid} \\
& = 2 \text{ if invalid function} \\
& = 0 \text{ if successful bcall} \\
\text{On successful bcall} \\
\text{state} & = 0 \text{ video disabled} \\
& = 1 \text{ video enabled}
\end{align*} \]

Explanation

Determines whether the video is enabled, setting 'state' appropriately.
This subsystem handles calls to the Parallel Port Printer driver.

**Supported Function Calls**

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read UART Configuration</td>
<td>13</td>
</tr>
</tbody>
</table>
SUBSYSTEM: PR DRIVER

FUNCTION: Read PR Configuration

Call

\[ ax = 13 \]  \text{(subsystem number)}

\[ si = 13 \]  \text{(function code)}

\[ es:bx = \text{parameter block address} \]

\[ di = 8 \]  \text{(size of parameter block in bytes)}

Parameter Block

\begin{verbatim}
struct
   version dw (?) ; version number of the PR driver

\end{verbatim}

Return

\[ bx = 1 \text{ if subsystem invalid} \]

\[ = 2 \text{ if invalid function} \]

\[ = 0 \text{ if successful bcall} \]

On successful bcall

Parameter Block = current PR configuration.

Explanation

Reads the current PR configuration into the parameter block.
## CONTENTS

<table>
<thead>
<tr>
<th>APPENDICES</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A - Clock Definitions</td>
<td>1</td>
</tr>
<tr>
<td>B - Floppy Disk Definitions</td>
<td>2</td>
</tr>
<tr>
<td>C - Video/LCD Definitions</td>
<td>3</td>
</tr>
<tr>
<td>D - Uart Definitions</td>
<td>5</td>
</tr>
<tr>
<td>E - Keyboard Definitions</td>
<td>6</td>
</tr>
<tr>
<td>F - ANSI Control Sequences</td>
<td>7</td>
</tr>
<tr>
<td>G - Lear Siegler Control Sequences</td>
<td>9</td>
</tr>
<tr>
<td>H - Programming Function Keys</td>
<td>11</td>
</tr>
</tbody>
</table>
Appendix A — Clock Driver

ALARM Information

ALARM struc
day dw (?) ; days since 1/1/30
min db (?) ; minutes
hour db (?) ; hours
hsec db (?) ; hundredths of sec
sec db (?) ; seconds
type db (?) ; alarm type
filler db (?) ; reserved
sflglp dd (?) ; alarm sleep flag segment-offset

addr
beep db (?) ; 1 - noisy, 0 - silent
srcid db (?) ; alarm id field

ALARM ends

TYPE field in ALARM structure

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>midnight alarm - for housekeeping</td>
</tr>
<tr>
<td>'D'</td>
<td>alarm occurs daily</td>
</tr>
<tr>
<td>'H'</td>
<td>alarm occurs hourly</td>
</tr>
<tr>
<td>'M'</td>
<td>alarm occurs every minute</td>
</tr>
<tr>
<td>'S'</td>
<td>alarm occurs every second</td>
</tr>
<tr>
<td>'T'</td>
<td>alarm is rm'd after 1st occurrence</td>
</tr>
<tr>
<td>'W'</td>
<td>alarm will restore power to off mc</td>
</tr>
</tbody>
</table>

SRCID field in ALARM structure

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'S'</td>
<td>system sourced alarm</td>
</tr>
<tr>
<td>'P'</td>
<td>magplanner alarm</td>
</tr>
</tbody>
</table>

SFLGLP field in ALARM structure

Valid states for the location pointed at by the SFLGLP field in the ALARM structure — only if the variable is in state PSLEEP will it be prodded to PWOKEN when the alarm triggers.

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSLEEP = 0x7073</td>
<td>process waiting for alarm</td>
</tr>
<tr>
<td>PWOKEN = 0x6f77</td>
<td>process has been woken</td>
</tr>
</tbody>
</table>

Maximum Number of Alarms 10
Appendix B – Floppy Disk

Bios Parameter Block Information

BFB struc
bp secss dw (?) ; sizeof sector, in bytes
bp clus db (?) ; sects per allocation unit
bp ressec dw (?) ; no of reserved sectors
bp ftcnt db (?) ; no of fats
bp dirent dw (?) ; no of entries in root dir
bp seccnt dw (?) ; total no of sects on disk
bp media db (?) ; media descriptor
bp ftsec dw (?) ; no of sects to each fat
BFB ends

FDBFB struc
fd bpb db sizeof BFB (?)
fd nosides db (?)
fd density db (?)
fd secptrk dw (?)
fd ltrack dw (?)
fd medsz db (?)
FDBFB ends

Read Write Disk Return Status

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>device is write protected</td>
</tr>
<tr>
<td>2</td>
<td>device not ready</td>
</tr>
<tr>
<td>4</td>
<td>bad command type</td>
</tr>
<tr>
<td>8</td>
<td>sector not found</td>
</tr>
<tr>
<td>10</td>
<td>write fault</td>
</tr>
</tbody>
</table>
Appendix C

General Screen Information

VIMOD struc

- **vm ansi**  
  - db (?); 1 implies ansi format
- **vm chprw**  
  - db (?); no of characters per row
- **vm rwpsn**  
  - db (?); no of rows per screen
- **vm dspen**  
  - db (?); 1 if display enabled
- **vm hscl**  
  - db (?); horizontal scaling factor
- **vm vscl**  
  - db (?); vertical scaling factor
- **vm curx**  
  - dw (?); current x position
- **vm cury**  
  - dw (?); current y position
- **vm curstr**  
  - db 8 dup (?); cursor mode
- **vm atr**  
  - db (?); attribute mode

VIMOD ends

Cursor Modes in VIMOD structure

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>cursor off</td>
</tr>
<tr>
<td>'b'</td>
<td>steady block cursor</td>
</tr>
<tr>
<td>'B'</td>
<td>blinking block cursor</td>
</tr>
<tr>
<td>'u'</td>
<td>steady underline cursor</td>
</tr>
<tr>
<td>'U'</td>
<td>blinking underline cursor</td>
</tr>
</tbody>
</table>

Display mode attributes in VIMOD structure

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>all attributes off</td>
</tr>
<tr>
<td>1</td>
<td>bold characters</td>
</tr>
<tr>
<td>4</td>
<td>underscore on (b/w displays)</td>
</tr>
<tr>
<td>5</td>
<td>blinking characters</td>
</tr>
<tr>
<td>7</td>
<td>reverse video</td>
</tr>
<tr>
<td>3</td>
<td>no reverse video</td>
</tr>
</tbody>
</table>
Specific LCD information

LCSCRN struc
sd smode db (?); current mode of LCD
sd wrap db (?); autowrap wrap/nowrap
sd wrping db (?); currently line wrapping
sd bposn dw (?); screen buffer position
LCSCRN ends

Each bit in the screen mode byte in the LCSCRN structure is individually defined as shown below from the high order bit to the low order bit.

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,6</td>
<td>0,0</td>
<td>Never set</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>Display off</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Always set</td>
</tr>
<tr>
<td>3,2</td>
<td>1,1</td>
<td>Character mode</td>
</tr>
<tr>
<td></td>
<td>0,1</td>
<td>Steady Underline Cursor</td>
</tr>
<tr>
<td></td>
<td>0,0</td>
<td>Cursor off</td>
</tr>
<tr>
<td></td>
<td>0,1</td>
<td>Blinking Block Cursor</td>
</tr>
<tr>
<td></td>
<td>0,0</td>
<td>Steady Block Cursor</td>
</tr>
</tbody>
</table>

Wrap mode definitions in LCSCRN structure

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>'w'</td>
<td>automatic line wrap on</td>
</tr>
<tr>
<td>'n'</td>
<td>automatic line wrap off</td>
</tr>
</tbody>
</table>

Cursor Position Structure

CPOS struc
dx coord dw (?); 0-(chars per line - 1)
y coord dw (?); 0-(lines per screen - 1)
CPOS ends

Screen Format Structure

SFRMT struc
sf lps dw (?); lines per screen
sf cpl dw (?); characters per line
SFRMT ends
### Parity Definitions

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>odd parity</td>
</tr>
<tr>
<td>1</td>
<td>parity disabled</td>
</tr>
<tr>
<td>2</td>
<td>even parity</td>
</tr>
</tbody>
</table>

### Data Bits Definitions

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>char len 5 stop bits 1.0</td>
</tr>
<tr>
<td>4</td>
<td>char len 5 stop bits 1.5</td>
</tr>
<tr>
<td>8</td>
<td>char len 6 stop bits 1.0</td>
</tr>
<tr>
<td>12</td>
<td>char len 6 stop bits 2.0</td>
</tr>
<tr>
<td>16</td>
<td>char len 7 stop bits 1.0</td>
</tr>
<tr>
<td>20</td>
<td>char len 7 stop bits 2.0</td>
</tr>
<tr>
<td>24</td>
<td>char len 8 stop bits 1.0</td>
</tr>
<tr>
<td>28</td>
<td>char len 8 stop bits 2.0</td>
</tr>
</tbody>
</table>

### Baudrate Definitions

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19200 baud</td>
</tr>
<tr>
<td>1</td>
<td>9600 baud</td>
</tr>
<tr>
<td>2</td>
<td>4800 baud</td>
</tr>
<tr>
<td>3</td>
<td>2400 baud</td>
</tr>
<tr>
<td>4</td>
<td>1200 baud</td>
</tr>
<tr>
<td>5</td>
<td>600 baud</td>
</tr>
<tr>
<td>6</td>
<td>300 baud</td>
</tr>
<tr>
<td>7</td>
<td>150 baud</td>
</tr>
</tbody>
</table>
Appendix D - UART Definitions

### Parity Definitions

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>odd parity</td>
</tr>
<tr>
<td>1</td>
<td>parity disabled</td>
</tr>
<tr>
<td>2</td>
<td>even parity</td>
</tr>
</tbody>
</table>

### Data Bits Definitions

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>char len 5 stop bits 1.0</td>
</tr>
<tr>
<td>4</td>
<td>char len 5 stop bits 1.5</td>
</tr>
<tr>
<td>8</td>
<td>char len 6 stop bits 1.0</td>
</tr>
<tr>
<td>12</td>
<td>char len 6 stop bits 2.0</td>
</tr>
<tr>
<td>16</td>
<td>char len 7 stop bits 1.0</td>
</tr>
<tr>
<td>20</td>
<td>char len 7 stop bits 2.0</td>
</tr>
<tr>
<td>24</td>
<td>char len 8 stop bits 1.0</td>
</tr>
<tr>
<td>28</td>
<td>char len 8 stop bits 2.0</td>
</tr>
</tbody>
</table>

### Baudrate Definitions

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19200 baud</td>
</tr>
<tr>
<td>1</td>
<td>9600 baud</td>
</tr>
<tr>
<td>2</td>
<td>4800 baud</td>
</tr>
<tr>
<td>3</td>
<td>2400 baud</td>
</tr>
<tr>
<td>4</td>
<td>1200 baud</td>
</tr>
<tr>
<td>5</td>
<td>600 baud</td>
</tr>
<tr>
<td>6</td>
<td>300 baud</td>
</tr>
<tr>
<td>7</td>
<td>150 baud</td>
</tr>
</tbody>
</table>
Appendix E - Keyboard Definitions

Special Keyboard Control Sequences

Some keyboard characters when typed from the command level of MS-DOS are not actually seen by the operating system but perform a specific BIOS function. These keys are listed below.

- `<CTRL>` <  Decrease LCD contrast.
- `<CTRL>` >  Increase LCD contrast.
- `<SHIFT>` <CTRL> ?  Toggle keyclick on/off.
- `<LOCK>`  Toggle shift lock on/off.
- `<SHIFT>` <CTRL> S  Enable 3 minute shutdown.
- `<SHIFT>` <CTRL> C  Disable 3 minute shutdown.
- `<SHIFT>` <CTRL> P  Terminate 'suspend' condition if data is directed to device PRN when no parallel printer is connected.

All other combinations of `<SHIFT>` <CTRL> <CHAR> are ignored.

Keyboard Codes for Function Keys

The keyboard has 12 special function keys labeled 'F1' to 'F12' and 4 arrow function keys labeled appropriately. Each of these 16 keys can be combined with the 'SHIFT' or 'CTRL' key to produce a total of 36 differently coded function keys. When any of the 36 combinations of keys are pressed a 2 byte code is returned. The first byte is always a null. The second byte uniquely defines one of the combinations as shown below.

<table>
<thead>
<tr>
<th>Function Key</th>
<th>Alone</th>
<th>SHIFT</th>
<th>CTRL</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>59</td>
<td>84</td>
<td>104</td>
</tr>
<tr>
<td>F2</td>
<td>60</td>
<td>85</td>
<td>105</td>
</tr>
<tr>
<td>F3</td>
<td>61</td>
<td>86</td>
<td>106</td>
</tr>
<tr>
<td>F4</td>
<td>62</td>
<td>87</td>
<td>107</td>
</tr>
<tr>
<td>F5</td>
<td>63</td>
<td>88</td>
<td>108</td>
</tr>
<tr>
<td>F6</td>
<td>64</td>
<td>89</td>
<td>109</td>
</tr>
<tr>
<td>F7</td>
<td>65</td>
<td>90</td>
<td>110</td>
</tr>
<tr>
<td>F8</td>
<td>66</td>
<td>91</td>
<td>111</td>
</tr>
<tr>
<td>F9</td>
<td>67</td>
<td>92</td>
<td>112</td>
</tr>
<tr>
<td>F10</td>
<td>68</td>
<td>93</td>
<td>113</td>
</tr>
<tr>
<td>F11</td>
<td>133</td>
<td>158</td>
<td>178</td>
</tr>
<tr>
<td>F12</td>
<td>134</td>
<td>159</td>
<td>179</td>
</tr>
<tr>
<td>Left Arrow</td>
<td>75</td>
<td>74</td>
<td>115</td>
</tr>
<tr>
<td>Right Arrow</td>
<td>77</td>
<td>78</td>
<td>116</td>
</tr>
<tr>
<td>Up Arrow</td>
<td>72</td>
<td>71</td>
<td>73</td>
</tr>
<tr>
<td>Down Arrow</td>
<td>80</td>
<td>79</td>
<td>81</td>
</tr>
</tbody>
</table>
Appendix F - ANSI Control Sequences

Introduction

Ansi sequences are character sequences which when output to a display device, in this case the character devices 'LCD' and 'VID', perform special functions.

All the control sequences are outlined below. Note that <ESC> represents the ESCAPE character (18 Hex or 27 Decimal), and that wherever a number appears in the sequence this is the ASCII representation for that number not the binary number itself.

**General Sequences**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ESC&gt;[0K</td>
<td>Clear screen to end of line from cursor position.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[2K</td>
<td>Clear entire line at cursor position.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[0J</td>
<td>Clear to end of screen from cursor position.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[2J</td>
<td>Clear entire screen.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[nnA</td>
<td>Move cursor up nn lines. For example &lt;ESC&gt;[11A moves the cursor up 11 lines and &lt;ESC&gt;[2A moves the cursor up 2 lines.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[nnB</td>
<td>Move cursor down nn lines.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[nnC</td>
<td>Move cursor right nn character positions.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[nnD</td>
<td>Move cursor left nn characters.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[yy;x</td>
<td>xH</td>
</tr>
</tbody>
</table>

**Setting Text Attributes (Video Only)**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ESC&gt;[0m</td>
<td>Enable normal video (ie white characters on a black background).</td>
</tr>
<tr>
<td>&lt;ESC&gt;[7m</td>
<td>Enable reverse video (ie black characters on a white background).</td>
</tr>
<tr>
<td>&lt;ESC&gt;[4m</td>
<td>Enable Underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[1m</td>
<td>Enable Highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[5m</td>
<td>Enable Blink.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[4;1m</td>
<td>Enable Underline highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[3;1m</td>
<td>Enable Blink highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[5;4m</td>
<td>Enable Blink underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[5;4;1m</td>
<td>Enable Blink underline highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[7;4m</td>
<td>Enable Reverse video underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[7;1m</td>
<td>Enable Reverse highlight.</td>
</tr>
</tbody>
</table>
Setting Text Attributes (Video Only) cont.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ESC&gt;[7;5m</td>
<td>Enable Reverse blink.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[7;4;1m</td>
<td>Enable Reverse underline highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[7;5;1m</td>
<td>Enable Reverse blink highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[7;5;4m</td>
<td>Enable Reverse blink underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[7;5;4;1m</td>
<td>Enable Reverse blink underline highlight.</td>
</tr>
</tbody>
</table>

Cursor Type Controls

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ESC&gt;[1v</td>
<td>Cursor off.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[2v</td>
<td>Cursor steady underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[3v</td>
<td>Cursor steady block. (Video only)</td>
</tr>
<tr>
<td>&lt;ESC&gt;[2v&lt;ESC&gt;[5v</td>
<td>Cursor blinking underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[3v&lt;ESC&gt;[5v</td>
<td>Cursor blink block.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[4v</td>
<td>Cursor steady.</td>
</tr>
<tr>
<td>&lt;ESC&gt;[5v</td>
<td>Cursor blink.</td>
</tr>
</tbody>
</table>
Appendix G - Lear Siegler Control Sequences

Introduction

There is available another standard of control sequences that perform a subset of the ANSI control sequences. This standard is called the Lear Siegler standard of cursor position and control.

The Lear Siegler sequences supported by the MAGNUM are illustrated below.

**Cursor Movement Control**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ESC&gt;M</td>
<td>Move cursor up a line.</td>
</tr>
<tr>
<td>&lt;ESC&gt;D</td>
<td>Move cursor down a line.</td>
</tr>
<tr>
<td>&lt;ESC&gt;E</td>
<td>Move cursor right a character position.</td>
</tr>
<tr>
<td>&lt;ESC&gt;=yx</td>
<td>Move cursor to line 'y' column 'x' where 'x' &amp; 'y' are characters obtained from the table below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CO-ORDINATE</th>
<th>CHARACTER</th>
<th>CO-ORDINATE</th>
<th>CHARACTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>' '</td>
<td>21</td>
<td>'4'</td>
</tr>
<tr>
<td>2</td>
<td>'!'</td>
<td>22</td>
<td>'5'</td>
</tr>
<tr>
<td>3</td>
<td>'&quot;'</td>
<td>23</td>
<td>'6'</td>
</tr>
<tr>
<td>4</td>
<td>'='</td>
<td>24</td>
<td>'7'</td>
</tr>
<tr>
<td>5</td>
<td>'$'</td>
<td>25</td>
<td>'8'</td>
</tr>
<tr>
<td>6</td>
<td>'%'</td>
<td>26</td>
<td>'9'</td>
</tr>
<tr>
<td>7</td>
<td>'&amp;'</td>
<td>27</td>
<td>';'</td>
</tr>
<tr>
<td>8</td>
<td>'*'</td>
<td>28</td>
<td>';'</td>
</tr>
<tr>
<td>9</td>
<td>'('</td>
<td>29</td>
<td>'&lt;'</td>
</tr>
<tr>
<td>10</td>
<td>')'</td>
<td>30</td>
<td>'='</td>
</tr>
<tr>
<td>11</td>
<td>'+'</td>
<td>31</td>
<td>'&gt;'</td>
</tr>
<tr>
<td>12</td>
<td>'+'</td>
<td>32</td>
<td>'1'</td>
</tr>
<tr>
<td>13</td>
<td>'-'</td>
<td>33</td>
<td>'2'</td>
</tr>
<tr>
<td>14</td>
<td>'-'</td>
<td>34</td>
<td>'3'</td>
</tr>
<tr>
<td>15</td>
<td>'.'</td>
<td>35</td>
<td>'4'</td>
</tr>
<tr>
<td>16</td>
<td>'/'</td>
<td>36</td>
<td>'5'</td>
</tr>
<tr>
<td>17</td>
<td>'0'</td>
<td>37</td>
<td>'6'</td>
</tr>
<tr>
<td>18</td>
<td>'1'</td>
<td>38</td>
<td>'7'</td>
</tr>
<tr>
<td>19</td>
<td>'2'</td>
<td>39</td>
<td>'8'</td>
</tr>
<tr>
<td>20</td>
<td>'3'</td>
<td>40</td>
<td>'9'</td>
</tr>
</tbody>
</table>

*(table continued over page)*
<table>
<thead>
<tr>
<th>CO-ORDINATE</th>
<th>CHARACTER</th>
<th>CO-ORDINATE</th>
<th>CHARACTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>'H'</td>
<td>61</td>
<td>'Y'</td>
</tr>
<tr>
<td>42</td>
<td>'I'</td>
<td>62</td>
<td>'J'</td>
</tr>
<tr>
<td>43</td>
<td>'J'</td>
<td>63</td>
<td>'L'</td>
</tr>
<tr>
<td>44</td>
<td>'K'</td>
<td>64</td>
<td>'M'</td>
</tr>
<tr>
<td>45</td>
<td>'L'</td>
<td>65</td>
<td>'N'</td>
</tr>
<tr>
<td>46</td>
<td>'M'</td>
<td>66</td>
<td>'O'</td>
</tr>
<tr>
<td>47</td>
<td>'N'</td>
<td>67</td>
<td>'P'</td>
</tr>
<tr>
<td>48</td>
<td>'O'</td>
<td>68</td>
<td>'Q'</td>
</tr>
<tr>
<td>49</td>
<td>'P'</td>
<td>69</td>
<td>'R'</td>
</tr>
<tr>
<td>50</td>
<td>'Q'</td>
<td>70</td>
<td>'S'</td>
</tr>
<tr>
<td>51</td>
<td>'R'</td>
<td>71</td>
<td>'T'</td>
</tr>
<tr>
<td>52</td>
<td>'S'</td>
<td>72</td>
<td>'U'</td>
</tr>
<tr>
<td>53</td>
<td>'T'</td>
<td>73</td>
<td>'V'</td>
</tr>
<tr>
<td>54</td>
<td>'U'</td>
<td>74</td>
<td>'W'</td>
</tr>
<tr>
<td>55</td>
<td>'V'</td>
<td>75</td>
<td>'X'</td>
</tr>
<tr>
<td>56</td>
<td>'W'</td>
<td>76</td>
<td>'Y'</td>
</tr>
<tr>
<td>57</td>
<td>'X'</td>
<td>77</td>
<td>'Z'</td>
</tr>
<tr>
<td>58</td>
<td>'Y'</td>
<td>78</td>
<td>'a'</td>
</tr>
<tr>
<td>59</td>
<td>'Z'</td>
<td>79</td>
<td>'b'</td>
</tr>
<tr>
<td>60</td>
<td>'a'</td>
<td>80</td>
<td>'c'</td>
</tr>
</tbody>
</table>

**Setting Text Attributes (Video Only)**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;ESC&gt;G0</td>
<td>Set normal video (ie white characters on a black background).</td>
</tr>
<tr>
<td>&lt;ESC&gt;G1</td>
<td>Set reverse video (ie black characters on a white background).</td>
</tr>
<tr>
<td>&lt;ESC&gt;G2</td>
<td>Enable Underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G3</td>
<td>Enable Highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G4</td>
<td>Enable Blink.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G5</td>
<td>Enable Underline highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G6</td>
<td>Enable Blink highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G7</td>
<td>Enable Blink underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G8</td>
<td>Enable Blink underline highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G9</td>
<td>Enable Reverse video underline.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G:</td>
<td>Enable Reverse highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G;</td>
<td>Enable Reverse blink.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G&lt;</td>
<td>Enable Reverse underline highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G=</td>
<td>Enable Reverse blink highlight.</td>
</tr>
<tr>
<td>&lt;ESC&gt;G&gt;</td>
<td>Enable Reverse blink underline.</td>
</tr>
</tbody>
</table>
Appendix K - Programming The Function Keys

The function keys on the internal keyboard may be programmed by ANSI control sequences sent to an an ANSI device i.e. 'LCD' or 'VID'. Each function key may have a definition of maximum length 40 characters with a total user space of 256 characters. Both <SHIFT> and <CTRL> function keys and <HELP> may be programmed giving a total of 39 programmable keys.

The sequences listed below outline how this feature is used.

Sequence
-------
\textless{}ESC\textgreater{}\{\textasciitilde{}nnn\textasciitilde{}R\textasciitilde{}\}---- string ----'p
\textless{}ESC\textgreater{}\{\textasciitilde{}nnn\textasciitilde{}R\textasciitilde{}A\textasciitilde{}A\textasciitilde{};B\textasciitilde{}B\textasciitilde{};C\textasciitilde{}C\textasciitilde{};D\textasciitilde{}p
\textless{}ESC\textgreater{}\{\textasciitilde{}nnn\textasciitilde{}R\textasciitilde{}\}---- string ----';A\textasciitilde{}A\textasciitilde{};B\textasciitilde{}p
\textless{}ESC\textgreater{}\{\textasciitilde{}nnn\textasciitilde{}R\textasciitilde{}A\textasciitilde{}A\textasciitilde{}B\textasciitilde{}B\textasciitilde{}C\textasciitilde{}C\textasciitilde{}\}---- string ----'p

Function
-------
to define a string of ASCII characters.
to define a string of decimal bytes.
to define a string of ASCII characters followed by decimal bytes (eg. 13 for <RETURN> and 10 for <LF>),
to define decimal bytes followed by an ANSI string.

NB: In the above sequences 'nnn' corresponds to 3 ASCII numbers representing the function key to be programmed. The first (leftmost) digit defines the normal/shift/ctrl key as follows

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal function key</td>
<td>Shifted function key</td>
<td>Ctrl'd function key</td>
<td>Shifted ctrl'd function key</td>
</tr>
</tbody>
</table>

The second and third digits specify the function key number to be programmed.

For example: 001 or just 1 corresponds to function key <F1>.
110 corresponds to <SHIFT> <F10>.
213 corresponds to <CTRL> <HELP>. 
Appendix I - Executing Programs from Rom

General Programming Techniques

Rom drives 'B:' and 'C:' are made to appear as standard block devices to Msdos by the Magnum BIOS software. Each Rom begins at segment address OCOOH in the processor map, and extends to OEOOH; only one module is enabled into the processor memory space at any one time, the switching being accomplished by a control register in processor I/O space at location O50H. A '0' selects the left hand module; '1' selects the right. Whilst selected, the Rom module acts as normal memory; and programs can be executed from it, with the obvious restriction that data cannot meaningfully be written to Rom; any data segments must be located down in the Magnum's Ram areas. This has the following implications -

(i) The Rom module must contain solely read-only information i.e. code and non-modifiable data.

(ii) Because of the mechanisms used by Msdos, the data segment cannot normally contain initialised data. All data must be explicitly initialised by the program.

Using the Msdos debugger, examine the MagCalc or MagWriter Rom module, and notice that each has two files, 'MX.EXE' and 'MX.ROM'. 'MX.EXE' is a 'loader' file; it contains only a long jump to the program stored in the '.ROM' file. When you execute 'MX' from the keyboard, the following sequence of events occurs -

(i) 'MX.EXE' is loaded into system Ram by Msdos. As part of the normal loading process for a '.EXE' file, all segment registers are set up. In particular, the 'DS:' register now points into Ram at the Program Description Area. This enables the program access to information about all the Ram space available to it. Consult the Msdos Programmer's Manual for a fuller explanation.

(ii) The long jump that forms the only code in 'MX.EXE' is executed, thus transferring control to the first byte of the 'MX.ROM' file in Rom. Note that this process is completely invisible to Msdos, which still believes it is executing the '.EXE' file. Note also that the Rom module containing the '.EXE' file is left selected by the Magnum BIOS Rom driver; thus there is no need for the '.EXE' program to explicitly select it.

In order to code the jump instruction in the '.EXE' file, the physical address of the start of the '.ROM' file in the module must be known. As described in the section of this manual on the Rom driver, the Rom is laid out like a standard Msdos disk; with FAT and directory sectors at the start followed by the files themeselves. Normally, the '.ROM' file is written to the
disk first, so that it is simple to locate; in this case it will begin immediately after the last directory sector.

The '.EXE' file must declare a stack segment, even if the '.ROM' file immediately re-defines it. This is because Msdos uses the program's stack when calling it; if the stack segment is not declared, results are undefined.

Software destined for the Rom modules may be developed on the Ram drive, 'D:;', only if it is formatted in a manner identical to that of the Rom modules. Support software for this is supplied along with the Eprom burner support disk, and is described in that product's user manual. In brief the process is as follows —

(i) Drive 'D:' is reformatted to the same size and layout as the target Rom.
(ii) '.ROM' and '.EXE' files are copied in order onto 'D:'
(iii) The burner program is run to copy the information onto the Rom[es].

If a program grows above the size that physically be fitted into a single module, it is feasible to split it between the two modules. As has been described, the two modules occupy the same physical memory space; when one is selected the other is inaccessible. The '.EXE' file must therefore provide some switching mechanism to enable program control transfers between the two modules. In the simplest case, for example, the '.EXE' file could supply a routine that selects the other Rom and then makes a far call to some defined entry point. On return it merely selects the original Rom and makes a far return.