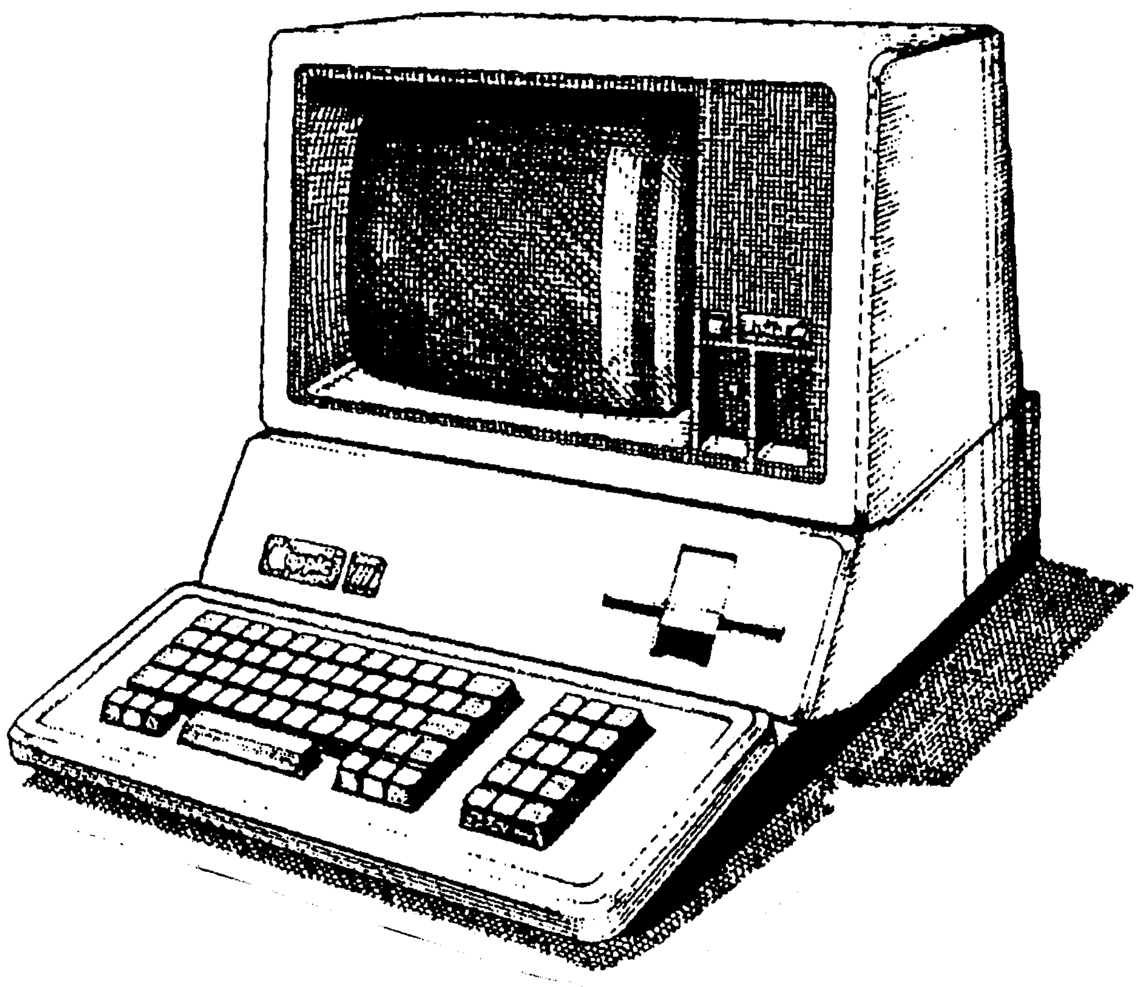
Apple /// Computer Information



DOCUMENT NAME

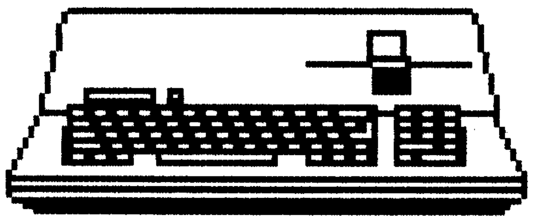
UNDOCUMENTED APPLE /// SOS FEATURES 1989

161

Ex Libris David T. Craig

"_113.PICT" 240 KB 2001-08-28 dpi: 300h x 300v pix: 2163h x 2892v

16 pages



Apple /// SOS 1.3
Technical Information

Undocumented Apple /// SOS Features

Written by Rick Sidwell • January 1989

Undocumented Apple /// SOS Features

Rick Sidwell (January 1989)

One of the nicest features of the Apple /// Computer is its operating system, SOS. In this "Sophisticated Operating System," Apple placed many routines to make it easy to write programs which are device independent and use the Apple ///'s memory in a cooperative manner. Most of the features of SOS are documented in three manuals: the SOS Reference Manual, Volumes 1 and 2, and the SOS Device Writer's Guide. However, the documentation for some of the features of SOS seems to have been "forgotten" by Apple; for one reason or another, the manuals don't tell the whole story. This article is about some of these undocumented features.

There are two types of SOS features which are not documented: features which should have been documented and items which were best left undocumented. The main reason that many things were left undocumented by Apple was to allow for upgrades. Modifications to SOS to fix bugs or add features would change the locations and possibly the meanings of many internal variables. For example, if a program depended on the fact that SOS keeps its Volume Control Blocks (VCB's) at locations $1000-$10FF, and that the device number is in the sixteenth byte of a VCB, that program would fail miserably if a new version of SOS changed the format of its VCB's or moved them to a different location. To allow flexibility for future versions of SOS, Apple did not document such things, and this article will not do so either.

However, there are a number of SOS calls, entry points, and global variables which seem to be independent of the SOS version, yet still not documented; these items are the subject of this article. There are two undocumented SOS calls, D_READ and D_WRITE, which allow you to read and write to disks without using the SOS file system. In addition to the normal SOS calls, the first half of page $19 contains some global variables and SOS entry points, only some of which have been documented. The undocumented entry points deal with NMI handling, buffer management, system failures, and clearing a file's backup bit. The global variables are for communication between SOS, device drivers, and the interpreter.

Direct Device I/O

SOS provides a sophisticated (by 1980 standards) file system which is sufficient for most needs. It is designed to be safe, so that you won't accidently trash a disk using file system commands. Because of this, there are some things that it can not do-- more dangerous commands are needed. Although not documented in the SOS manuals, SOS does provide these dangerous commands: Device calls $80 and $81 are D_READ and D_WRITE, which call a device driver to read and write directly, bypassing the SOS file system.

If the SOS Reference manual included a description of D_READ and D_WRITE, it would certainly have been accompanied by a stern warning such as the following (adapted from the ProDOS manual):

> **WARNING**: These commands are intended to be used by utility and diagnostic programs. Application programs should not use these commands: they can very easily damage the integrity of the SOS file structure. All necessary functions can be performed without these commands.

Consider yourself warned! However, if you are writing a utility or diagnostic program, the D_READ and D_WRITE commands are used like any other SOS command; the process is decribed in detail in chapter 8 of the SOS reference manual. You issue a BRK command, followed by the command number (which is $80 for D_READ and $81 for D_WRITE), followed by the address of the parameter list:

```
BRK
.BYTE   80      ;for D_Read (use 81 for D_Write)
.WORD   DRLIST
```

The parameter lists have the following format:

```
             D_READ  $80   D_WRITE $81
             -----------   -----------
0            |   $05   |   |   $04   |      # parameters
             |---------|   |---------|
1            | dev_num |   | dev_num |      device number
             |---------|   |---------|
2            | buffer  |   | buffer  |      buffer pointer
3            |         |   |         |
             |---------|   |---------|
4            | request |   | request |      number of bytes
5            |  count  |   |  count  |      to read or write
             |---------|   |---------|
6            |  block  |   |  block  |      Block number
7            |         |   |         |
             |---------|   -----------
8            | transfer|                    number of bytes
9            |  count  |                    actually read
             -----------
```

A description of each parameter follows, in a style similar to the SOS Reference manual.

dev_num: 1 byte value
    Range: $01..$18

This is the device number of the device to read or write, obtained from the GET_DEV_NUM call.

buffer: pointer

This is a pointer to the buffer to which the data will be read (or from which it will be written). It must be large enough to hold the requested number of bytes. As with all SOS pointers, it can contain the address of either an absolute location in the current bank, or a zero page location which contains an extended address in any bank.

request_count: value
    Range: $0000..$FFFF

This is the number of bytes to read or write. For block devices, it must be a multiple of 512 ($200) or a BYTECNT error will occur.

block: value
    Range: $0000..$FFFF

This the block number of the first block to be transferred. It is ignored for character devices. For block devices, the blocks are numbered from $0000 to the highest block on the disk (one less than the number of blocks).

transfer_count: 2 byte result
Range: $0000..$FFFF

If a D_READ is successful, the number of bytes actually read is returned in this parameter. It will be less than request_count if the newline character is encountered when reading from a character device with newline mode set.

Errors

$11:	BADDNUM	Invalid device number
$23:	NOTOPEN	Device not open
$26:	BADOP	Invalid operation
$27:	IOERROR	I/O error
$28:	NODRIVE	Drive not connected
$2A:	CRCERR	Checksum error
$2B:	NOWRITE	Device write protected
$2C:	BYTECNT	Byte count not multiple of 512
$2D:	BLKNUM	Block number too large
$2E:	DISKSW	Disk switched
$30..$3F:		Device-specific errors

Before leaving the D_READ and D_WRITE commands, consider the following situations when you should NOT use them. If you are using a character device, you should use the file READ and WRITE commands; although D_READ and D_WRITE do almost exactly the same thing, if you accidently do a D_WRITE to a block device instead of a character device, you will probably inadvertantly destroy some data. If you need to read a directory, use OPEN and READ rather than D_READ; it is easier since SOS already has the code to figure out where the directory is located. If you need to write to a directory, be careful! SOS can modify most attributes with the RENAME or SET_FILE_INFO commands, and is very careful that the correct disk is in the drive before it does so.

Finally, here are a few of the situations for which the D_READ and D_WRITE commands are essential. Reading and writing the boot blocks (blocks 0 and 1 of the disk), as well as putting a blank directory on a freshly formatted disk are impossible to do with SOSmmands. A program which verifies the file structure of a questionable disk to make sure that all of the pointers and bitmap are consistent would have to use D_READ; the index blocks are not accessible using SOS file commands. An important application of D_READ and D_WRITE is to access files on non-SOS disks; the Apple /// Pascal system, for example, uses them to access Apple][Pascal disks.

As an example of using D_READ and D_WRITE, a simple interpreter which copies the contents of the disk in drive 1 to the disk in drive 2 is in a separate file called DCOPY.ASM. For simplicity, many features which should be present in such a program, such as verifying the copy and nice error handling, have been left out.

Non-Maskable Interrupts

The Apple /// supports several kinds of interrupts. The "normal" interrupts are called IRQ's (which means interrupt request), and are handled by SOS using the SIR mechanism described in the SOS Device Driver Writer's Guide. Another kind of interrupt is the Non-Maskable Interrupt, or NMI. This name comes from the fact that there is a machine language instruction which will mask (or disable) IRQ interrupts, but NMI interrupts can not be masked. They are generally used for things which are so important that they should not be ignored. On the Apple ///, NMI interrupts can be generated from two sources: a peripheral card in one of the I/O slots and the Reset key on the keyboard. They are handled completely differently.

Slot NMI's

Peripheral card devices which are interrupt driven should generally use IRQ interrupts rather than NMI interrupts. There are four reasons for this: First, only one device may use NMI interrupts at any given time; if two devices are configured which both depend on NMI interrupts, only one may be opened at a time. Second, critical sections of code can temporarily mask IRQ's to prevent them from being interrupted; although NMI's can be disabled, it is not as convenient. Third, NMI handlers may not call ANY SOS routines, including such useful ones as QUEEVENT and SELC800. Fourth, if an interpreter disables the Reset key, either as part of a copy-protection scheme or for some other reason, NMI interrupts will also be disabled. Thus NMI driven devices can not be used with such interpreters as Apple Writer.

Although it should be avoided, using NMI interrupts is much faster than using IRQ interrupts. Besides the faster dispatch time (SOS does not need to poll each device to determine which device caused the interrupt), NMI interrupts have a higher priority than any IRQ interrupt, and will be acknowledged even if interrupts are disabled while executing an IRQ handler. To use NMI interrupts effectively, the device driver should be able to anticipate when the device will generate an interrupt. For example, a disk drive which will interrupt shortly after an I/O request is given would work, while a serial port which might generate an interrupt at unpredictable times would not. If the NMI interrupt can be anticipated, then SOS will wait for the interrupt before granting a user request to disable NMI interrupts.

A device driver can determine whether NMI interrupts are enabled or not by examining bit 4 of the environment register ($FFDF). If it is set, NMI's are enabled; if it is reset, they are not. A device driver should not set this bit, but simply report an error if NMI's are disabled and it needs them. A device driver reserves the NMI interrupt in the same way as it does an IRQ interrupt: by using ALLOCSIR (JSR $1913) as described in the SOS Device Driver Writer's Guide; the SIR number for the NMI is 0. This is usually done during the device D_OPEN call for character devices or the D_INIT call for block devices.

When an I/O operation is requested which will cause an NMI interrupt, the driver should do the following: Make sure that NMI's are enabled (as described above), set bit 7 of NMIPend ($1903) to prevent them from being disabled before the NMI occurs, and perform the initial operations on the device which can be done before the interrupt occurs. When the NMI occurs, the NMI handler should reset bit 7 of NMIPend($1903), and then handle the interrupt. If the interpreter should request SOS to disable NMI's by calling NMIDSBL while bit 7 of NMIPEND is set, SOS will wait until it is reset before disabling the NMI's. If the NMI does not occur within a short time, SOS will halt with system failure $04.

Keyboard NMI's

A keyboard NMI is generated when the user presses the Reset key (by itself; Control-Reset will reboot the system). Unlike slot NMI's, keyboard NMI's are handled by the interpreter. When SOS gets a keyboard NMI, it calls the routine at USRNMI ($1910); this is a JMP instruction which the interpreter can fill in with the proper address. Store the low byte in $1911 and the high byte in $1912 (location $1910 contains the JMP op-code). This address must be in the system bank (the one containing the interpreter). Should the interpreter need to restore the default handler (which does nothing), its address is stored at $1904 and $1905.

A keyboard NMI handler has the same restrictions as a slot NMI handler, namely NO SOS routines may be called. A typical use of the keyboard NMI is to set a flag which will cause an executing program to abort at a convenient time.

Disabling NMI's

Although NMI interrupts are not maskable with ordinary 6502 instructions, the Apple /// and SOS provide a way to disable them. The same mechanism disables slot NMI's, keyboard NMI's, and rebooting via Control-Reset; there is no way to disable one without disabling the others. Disabling NMI's is useful in several situations: Critical routines, such as those found in disk drivers, may disable NMI interrupts as well as IRQ interrupts both to prevent them from interfering with the timing and to prevent a reboot from destroying data. An interpreter might disable NMI's during critical operations such as a data base update to help maintain the data base integrity. A copy-protection scheme may disable NMI's to prevent hackers from using the Apple /// monitor to help break the protection.

SOS provides the following two entry points to disable and re-enable NMI interrupts:

NMIDSBL Entry point $1919

NMIDSBL will disable NMI interrupts and the Reset key. In order to allow pending NMI's to be serviced, NMIDSBL waits until bit 7 of NMIPEND ($1903) is clear as described above. If the NMI does not occur within a short period of time, system failure $04 will occur. NMIDSBL is normally called by the interpreter. Device drivers and interrupt handlers may call it if needed, however NMI's will be re-enabled when control is returned to the interpreter; only the interpreter may disable NMI's permanently.

NMIENBL Entry point $191C

NMIENBL will enable NMI interrupts and the Reset key. NMIENBL should be called only by an interpreter. Other environments may restore the NMI enable status after finishing a critical section protected by NMIDSBL by saving the environment register ($FFDF) before calling NMIDSBL, and restoring it after the critical section.

Buffers

Interpreters and user programs can request memory by using one of the memory manager SOS calls, but device drivers run in the SOS environment and can not perform SOS calls. Normally, a device driver will reserve enough memory in its code to suit its needs, but SOS does provide a method for device drivers to request extra memory using the same mechanism it uses to allocate buffers for file I/O.

Buffers are special in two ways. First, SOS may move buffers around in order to prevent memory fragmentation. Second, an integrity check is built into SOS to help detect unauthorized modification of buffers. Caution must be used with the arguments passed to these routines. An invalid buffer number or size is a FATAL error; the system will halt. It does this since buffers are used mostly by the file system, and errors there can easily destroy a disk--rather than risk this, SOS will halt the system if an error is detected.

There are three SOS routines which deal with buffers: REQBUF (JSR $192B), GETBUFADR (JSR $192E), and RELBUF (JSR $1931). Only a device driver may call these routines--interpreters and interrupt handlers may not use them. Furthermore, a device driver's D_INIT routine should not use these routines; for some reason SOS initializes the device drivers before it initializes either the buffer manager or the memory manager.

REQBUF Entry Point $192B

REQBUF is used to request a buffer from SOS. The buffer may be up to 64 pages long.

Parameters passed:

```
        A:      Size of the buffer in pages ($01..$40)
```

Normal exit:

```
        Carry:  Clear
        A:      Buffer number ($01..$1F)
```

Error exit:

```
        Carry:  Set
        A:      Error code
```

Errors:

```
$54:    OUTOFMEM          Memory manager could not allocate a buffer
$55:    BUFTBLFULL        Buffer table full
System failure $10:       Invalid buffer size
```

GETBUFADR Entry Point $192E

GETBUFADR is used to get the current address of a previously allocated buffer. Since buffers can move around in memory, you should call GETBUFADR each time your driver is called. GETBUFADR also makes sure that the first byte of the buffer has not changed since the last time you driver exited after using it; system failure $0F (invalid buffer number) will occur if it has.

The address of the buffer is placed in the zero page location stored in the X register. For example, if X contains $D4 when GETBUFADR is called, the buffer address will be placed in $D4 and $D5, with the X-byte in $14D5. No error return is possible; all errors are fatal.

Parameters passed:

```
        A:      Buffer number
        X:      Address of zero page location to put address
```

Normal exit:

```
        Carry:  Clear
        A:      Size of the buffer in pages
        (X):    The buffer address (in the specified location)
```

Errors:

```
System failure $0F:     Invalid buffer number
```

RELBUF Entry Point $1931

RELBUF releases a buffer allocated by REQBUF. It then moves some or all of the buffers to new memory locations to prevent the memory from becoming fragmented. You will rarely use more than one buffer; if you do, you will need to use GETBUFADR to get the addresses of the remaining buffers after releasing one of them with RELBUF. No non-fatal errors are possible.

Parameters passed:

```
        A:      Buffer number
```

Errors:

```
System Failure $10:     Invalid buffer number
```

Other SOS Entry Points

Most of the other SOS entry points are documented in chapter 4 of the SOS Device Driver Writer's Guide. The two that are not are described below:

SYSFAIL Entry point $1925

SYSFAIL is used when an irrecoverable error is encountered. Since any data not written to disk will be lost, this routine should be called only in dire emergency, when the alternative is worse. A list of the error numbers with a description of each error is given in Appendix D of the SOS Reference Manual.

SYSFAIL displays SYSTEM FAILURE and the error number in the bottom right corner of the screen and hangs the system. The only possible recovery is to reboot the system. However to help debugging, SYSFAIL stores the system status on page $19; these locations can be accessed with the Apple /// monitor. The following information is stored:

$19F6	Bank register
$19F7	Zero page register
$19F8	Environment register
$19F9	6502 Y register
$19FA	6502 X register
$19FB	System failure number
$19FC	6502 status register
$19FD/$19FE	Program counter
$19FF	Stack pointer
$1700-$17FF	System stack

Parameters passed:

A:	The system failure number

CLRBACK Entry point $1934

Every SOS file has a bit known as the backup bit. This bit is set whenever any operation is done which modifies the file. Backup programs, such as BACKUP /// can examine this bit to determine which files have been modified since the last backup. After backing up the file, the program can then clear the bit. Clearing the backup bit is done by calling CLRBACK just before calling SET_FILE_INFO; this forces SOS to use the setting given in the SET_FILE_INFO parameter list rather than automatically setting it. Note that

CLRBACK must be called just before each SET_FILE_INFO call which is to clear the backup bit.

Parameters passed:

```
        A:      $00 to use the given backup bit setting
                $20 to always set the backup bit
```

SOS Globals

SOS reserves several locations for communication between SOS, the interpreter, and device drivers. These locations are described below. They should never be modified except as indicated.

MEMSIZE Location $1900

MEMSIZE contains the size of the memory in 16K units. It is used to determine the size of the machine. A 128K system will have a MEMSIZE of $08, a 256K system will have a MEMSIZE of $10, and a 512K system will have a MEMSIZE of $20.

SYSBANK Location $1901

SYSBANK contains the bank number of the system bank, which happens to be the highest bank. It is used by SOS as the bank of the USRNMI routine, and can be used by an interpreter which uses overlays in different banks to switch back to the bank containing the main interpreter code.

SUSPFLSH Location $1902

This location is not used by SOS, but by the .CONSOLE driver for its suspend and flush output features (normally controlled by Control-7 and Control-9 on the keypad). Bit 7 is set if output is suspended; Control-7 will toggle it, and Control-9 will reset it. Bit 6 is set if output is being flushed; Control-9 will toggle it. The other bits are not currently used.

Although normally used only by the .CONSOLE driver, other programs may find SUSPFLSH useful also (be sure to disable interrupts while you modify a bit to preserve the integrity of the other bits). For example, a program can frustrate a user by resetting bit 7 just before doing any output (this prevents the user from stopping the output). A more useful program can examine bit 6 occasionally and abort the output if it is just being flushed. When a program reaches a point where the user would probably like to turn off flushing, it can do so automatically either by performing a Reset Driver D_CONTROL call (which has other effects, like clearing type-ahead), or by resetting bits 6 and 7 of SUSPFLSH.

NMIPEND Location $1903

Bit 7 of NMIPEND is set by device drivers which expect an NMI interrupt shortly, and reset by the NMI handler when it occurs. It is used to prevent the NMI from being disabled by a call to NMIDSBL until after the NMI has occurred. It should be modified only by a device driver which has successfully allocated SIR 0. It may be read by any routine, possibly to avoid a system failure caused by calling NMIDSBL if the NMI does not occur.

DFLTNMI Locations $1904-$1905

DFLTNMI contains the default keyboard NMI handler, which does nothing. It may be used by the interpreter to remove its own handler if it no longer wishes to handle keyboard NMI's.

SCRNMODE Location $1906

SCRNMODE stores the current mode of the Apple /// screen. It should be modified whenever the mode of the screen is changed. The previous value may be saved to restore the screen to the previous mode. For example, when SOS asks the user to change disks in some drive, it uses SCRNMODE to restore the screen to its previous setting when the disk is changed. SOS makes no attempt to ensure that the screen is in the correct mode except for the screen on/off bit, which it sets whenever it returns to the user after a SOS call or an interrupt.

The bits have the following meanings:

```
Bit 7:  0 for screen off, 1 for screen on
Bit 6:  0 for text, 1 for graphics
Bit 2:  0 for page 1, 1 for page 2
Bit 1:  0 for 40 columns, 1 for 80 columns
Bit 0:  0 for black and white, 1 for color
```

GRAFMEM Location $1907

GRAFMEM contains the number of pages currently reserved for graphics. Graphics memory starts at location $2000 of bank 0. Since device drivers, in particular the .GRAFIX driver, can not make SOS calls to reserve memory, it depends on the interpreter to reserve memory for the graphics screens. The interpreter should set this location accordingly when it does so. The .GRAFIX driver makes sure that the memory has been reserved before modifying it by drawing on a graphics screen.

DISKBUSY Location $1908

DISKBUSY is non-zero if a disk driver is accessing the disk. Although the driver takes care to disable interrupts when it is actually reading or writing data, it allows interrupts to occur between blocks. If these interrupts take too long, the next block may pass by the read/write head when the driver isn't looking. This is not serious--the block will come around again. But disk access will be slowed down. To prevent this, an interrupt handler which takes a relatively long time should check this byte, and if it is not zero, should avoid the long operation. If this is not possible, there is no harm done--just a longer disk access time.

Summary of SOS Globals and Entry Points

$1900	MEMSIZE	The size of the memory in 16K units.
$1901	SYSBANK	The system (highest) bank number
$1902	SUSPFLSH	Output suspend/flush flag Bit 7: 1 if scrolling is stopped Bit 6: 1 if output is disabled
$1903	NMIPEND	Bit 7 is set if an NMI is pending
$1904	DEFNMI (L)	Low byte of the default user NMI handler
$1905	DEFNMI (H)	High byte of the default user NMI handler
$1906	SCRNMODE	Screen mode flag Bit 7: 0 for screen off, 1 for on Bit 6: 0 for text, 1 for graphics Bit 2: 0 for page 1, 1 for page 2 Bit 1: 0 for 40 col., 1 for 80 Bit 0: 0 for b/w, 1 for color
$1907	GRAFMEM	Amount of memory allocated for graphics
$1908	DISKBUSY	Non-zero if a disk is busy
$1910	USRNMI	JMP to user's keyboard NMI handler
$1913	ALLOCSIR	Allocate an SIR
$1916	DEALCSIR	Deallocate an SIR
$1919	NMIDSBL	Disable NMI's and the Reset key
$191C	NMIENBL	Enable NMI's and the Reset key
$191F	QUEEVENT	Queue an event
$1922	SELC800	Select a slot's C800 ROM space
$1925	SYSFAIL	Report a system failure
$1928	SYSERR	Report an error
$192B	REQBUF	Request a buffer
$192E	GETBUFADR	Get the address of a buffer
$1931	RELBUF	Release a buffer
$1934	CLRBACK	Let SET_FILE_INFO clear the backup bit

<<< FINIS >>>