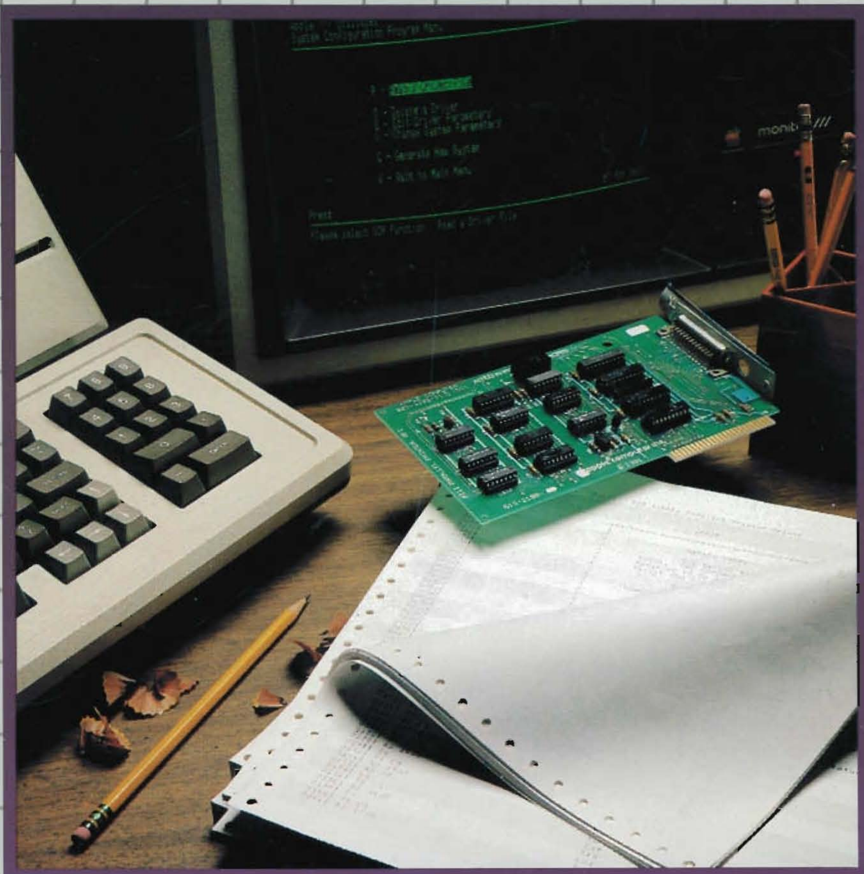
Apple ///

SOS

Device Driver Writer's Guide



Apple III

SOS Device Driver Writer's

Guide

Contents

2 The Physical Environment of SOS 13

3 Request Handling 23

4 SOS-provided Services 47

5 Interrupt Handling 59

6 Device Driver Coding Techniques 65

7 Interfacing with Apple III Peripheral Connectors 71

Appendices

Introduction

The device driver is an essential and integral part of the Apple III operating system, hereafter referred to as SOS (Sophisticated Operating System). It is the part of SOS that supports all input and output (I/O) operations, regardless of the type of device being used.

In the world of SOS, everything external to the CPU and its memory address space is a file: to be opened, read, written to, and closed. Unlike many other computer systems, the type of device being used for I/O makes essentially no difference in the way that programs perceive and use them.

Device drivers write to and read from files. This manual tells you how to write device drivers and incorporate them into SOS. It assumes that you are familiar with both 6502 assembly-language programming and the information in the following four manuals:

Apple III Owner's Guide
Apple III Standard Device Drivers Manual
Apple III SOS Reference Manual
Apple III Pascal Program Preparation Tools

If that assumption is not yet correct, we can resume when you return.

Why Device Drivers?

Most of us are used to speaking with people who use and understand the same language that we do. When someone new moves into the neighborhood speaking another language, we can either learn the new language, find a translater, wait for the other person to learn your language, or else get by without communicating.

A computer system is like a neighborhood, and each different device connected to the computer "speaks differently". If each application written to run on a computer is required to have its own routines to communicate with devices, a great amount of time (and money) is spent on needlessly duplicating effort. Rather than require users to write new interfacing programs or rewrite applications for each new device that they connect to their Apple III, SOS device drivers support uniform communication between applications and devices.

Device drivers become part of SOS and so are loaded each time the system is booted. *All* I/O in SOS is performed by device drivers.

Who Uses Them?

Every part of the Apple III system that communicates with something or someone external to the Apple III's processor uses device drivers in SOS, and no I/O is done without them. Some device drivers are supplied with SOS, including .CONSOLE, .PRINTER, .AUDIO, and .RS232 ; they are described in the *Apple III Standard Device Drivers Manual*.

Other device drivers are supplied with the device that they serve, for example .PROFILE, supplied with the ProFile hard disk.

How They Work

All SOS data flow is performed by device drivers through files. A file is a named, ordered sequence of bytes and may be used to store, transmit, or retrieve any type of information that you can put into the Apple III.

SOS recognizes two classes of files: character files and block files.

A character file is treated by SOS as an continuous stream of bytes. SOS can read or write the next byte in the stream, but it cannot reread or skip bytes in the stream.

A file sent to a character device, such as a printer, is a character device file. As far as a program running under SOS is concerned, there is no difference in the way it accesses any type of character device; all look like files to the program.

A file can also reside on a block device, such as a disk drive. A block file is composed of characters in groups called *blocks* of 512 bytes each. Blocks are numbered serially, but SOS can read from or write to any given block at will. A block file is limited to a maximum of $FFFFFE bytes, or 16,777,215 bytes.

A program can open, read, write, and close a character file, but cannot create, delete, or rename one. A character device file cannot be accessed as a random-access file; a block device file can be accessed randomly.

Scope of this Manual

This manual provides enough information for experienced assembly-language programmers to write device drivers for character and block devices to work with Apple III SOS.

This manual is not intended to be a tutorial covering basic programming or hardware-design techniques; we assume that you know them already.

Chapter 1 provides a general overview of the concepts underlying SOS device drivers.

Chapter 2 describes in general terms the underlying physical environment of SOS device drivers.

Chapter 3 describes request handling, the main "job" of device drivers.

Chapter 4 describes the services provided by SOS to aid device driver function, such as error reporting and resource allocation.

Chapter 5 describes interrupts and interrupt handling by SOS device drivers.

Chapter 6 presents techniques for developing device drivers.

Chapter 7 presents techniques for designing and building interface cards to connect with the Apple III through the backplane peripheral connectors.

Appendix A is a sample device driver skeleton that can be used as a starting point for writing drivers for block devices such as disks.

Appendix B is a sample device driver skeleton that can be used as a starting point for writing drivers for character devices such as printers.

Appendix C contains the instruction set of the 6502B, the microprocessor used by the Apple III.

Appendix D contains a list of system addresses that are important to device driver writers.

Apple II Emulation Mode

The Apple III also offers an Apple II Emulation mode. In this mode, the Apple III functions as a 48K Apple II or Apple II Plus with a disk controller card in slot 6, and a serial (either Communication or Serial) interface card in slot 5 or 7. There is no "slot 0". Other limitations of Emulation mode operation are:

- No software requiring the *Language card* will run on an Apple III in Emulation mode.

- Only the built-in disk drive and the first external drive will be usable. Daisy-chaining additional drives is not supported.
- The RGB video output will only generate black and white images in HIRES graphics.
- There is no cassette port.
- DMA and interrupts are not supported.

Notations Used in this Manual

Three symbols appear throughout this manual to point out particularly important information:

A hand indicates information of an especially useful nature, which may not be very obvious at first sight.

An eye points out some characteristic of the software or hardware operation that you should be careful about.

A stop sign draws your attention to something that may have serious consequences if not used properly, such as damaging the Apple III or causing a serious error, or complete shutdown of system operation.

1

Overview of SOS Device Drivers

1

Overview of SOS Device Drivers

The Apple III/SOS system deals with all input and output (I/O) in the same way: all devices connected to the system are files, communicating with SOS through device drivers.

Every device driver has one or more physical devices associated with it. For example, a block device driver has one or more block devices, a format device driver has one or more format devices, and so on.

SOS communicates to attached devices (keyboard, screen, printers, disks, and so on) by sending device requests to direct the operation of each device by its device driver. Remember that all devices connected to SOS are files.

A device driver is a memory-resident module that implements the set of SOS device requests (through request handlers) required of all devices connected to SOS. In addition to device requests, a device driver also performs interrupt handling (with interrupt handlers) for devices using interrupts.

At system startup, device drivers reside in a file called SOS.DRIVER on the boot volume. You can change the content of SOS.DRIVER with the SOS System Configuration Program (SCP) described in the *Apple III Standard Device Drivers Manual*. SCP lets you reconfigure your operating system by adding or removing device drivers. Note that SCP also checks the validity of your device driver's format.

When a device driver is called, the SOS device manager passes a request table to the device driver defining the type of operation to be done. These operations are called device requests, and each device driver has a specific set of device requests that it must perform for its own device. SOS device requests are briefly described later in this chapter, and in detail in Chapter 3.

A standard group of device drivers comes with every Apple III system to enable the operation of the Apple III's built-in devices, such as speaker, screen, keyboard, and RS232 serial port. These device drivers are described in the *Apple III Standard Device Drivers Manual*.

When you obtain an optional accessory device that can be connected to your Apple III, the device driver needed to operate it is also supplied.

Table 1-1 lists some important device drivers and the devices they serve.

Device Driver	**Device(s) Served**
(names as supplied)	
.CONSOLE	Screen and Keyboard
.PRINTER .RS232	Apple III serial port
.AUDIO	Apple III speaker
.GRAFIX	Apple III graphics display
.D1 through .D4	Disk III disk drives
.PROFILE	ProFile hard disk

Table 1-1. SOS Device Drivers and Devices

All the device drivers listed in Table 1-1 except .PROFILE and the Disk III drivers .D2 through .D4 operate built-in devices, and all except .PROFILE are supplied with the Apple III system software package. The .PROFILE driver is supplied with the ProFile hard disk, and is typical of device drivers supplied with Apple III optional devices. Its use is described in the documentation supplied with the ProFile hard disk.

SOS Device Classes

There are two classes of devices (and device drivers) within Apple III SOS: character devices and block devices.

Character devices, such as printers and modems, can transfer information in sequential character streams up to 64K bytes in length at one time.

Block devices, such as disks, transfer information in 512-byte blocks. Any higher orders of organization, such as files and directories, are the responsibility of SOS.

A subclass of the block device driver is the format driver, used to format a block device before use. A format device driver may either be part of a block device driver or stand alone. A format driver should be included as part of the device driver except when the format driver is very large. In such a case, memory limitations would dictate the need for a stand alone format driver.

Examples of stand alone format device drivers are .FMTD1 through .FMTD4, found on the SOS Utilities diskette and used by SCP to format diskettes.

Character Driver Functions

Character device drivers move character streams either in one direction, like .PRINTER, or bidirectionally, like .RS232. .

Character drivers must support NEWLINE mode. This allows the use of a single character to mark a logical end of record in a character stream. The NEWLINE character may be defined any number of times through DR_CONTROL device requests.

The SOS device requests performed by character device drivers are described briefly below, and in greater detail in Chapter 3. Device requests are issued by the SOS device manager.

DR_INIT

DR_INIT operates once only (during system startup) to prepare the device driver for use. The device served by the driver is not accessed and remains closed, and no resources are allocated.

DR_OPEN

DR_OPEN is called to allocate a resource from the system: in this case, to open its device file to be either written to or read from.

DR_CLOSE

DR_CLOSE is called to perform two operations: it shuts down its device, and it deallocates the system resources assigned to the driver and gives them back to the system.

DR_READ

DR_READ is called to read a specified number of characters from its character device into a buffer in memory.

DR_WRITE

DR_WRITE is called to write a specified number of characters from a buffer in memory out to the character device.

DR_STATUS

DR_STATUS is called to provide information on the current status of its device. In addition to the device's status, other information specific to a given device or driver may be returned.

DR_CONTROL

DR_CONTROL is called to reset the device, load control parameters, reset the NEWLINE character (described in Chapter 3), or make other changes to the device's operating parameters.

Block Driver Functions

Block devices move data in 512-byte blocks, and allow SOS to access easily any given logical block of a block device.

A block driver's device is divided into consecutively-numbered logical blocks; higher orders of organization (such as files or directories) on the device are handled outside the driver.

The SOS device requests implemented by block device drivers are briefly described below and in detail in Chapter 3.

DR_INIT

DR_INIT is called during system startup to perform operations required to prepare the device for use, allocate resources needed by the driver, and open the device. A DR_INIT request for a block device is equivalent to requesting DR_INIT and DR_OPEN for a character device.

DR_READ

DR_READ is called to read one or more blocks from the block device, beginning at a specified logical block number.

DR_WRITE

DR_WRITE is called to write a specified number of 512-byte blocks onto the block device from a buffer in memory, beginning at a given logical block number on the device.

DR_REPEAT

DR_REPEAT is called to repeat a DR_READ or DR_WRITE operation on a device. The unit number given for the call must be the same as the last unit called by the SOS device manager, and the last operation performed by that unit must have been DR_READ or DR_WRITE.

DR_STATUS

DR_STATUS is called by the SOS device manager to return the status of its block device. Either a status byte (whose format is defined in the driver's documentation), or the preferred location of a bitmap may be returned.

DR_CONTROL

DR_CONTROL is called to format the device.

Conceptual Model of SOS

It is often helpful for you to have a mental image of SOS and the relation of device drivers to it when you are creating a new driver.

The conceptual model of SOS presented below is purposely incomplete and slanted toward device drivers. The *Apple III SOS Reference Manual* gives a more complete picture, and you should understand it well before you begin writing device drivers.

The Abstract Machine

The Apple III/SOS system is defined in terms of an abstract machine whose operation and performance is a combination of the two parts of the system, SOS and the Apple III.

Figure 1-1 shows the components of the SOS abstract machine.

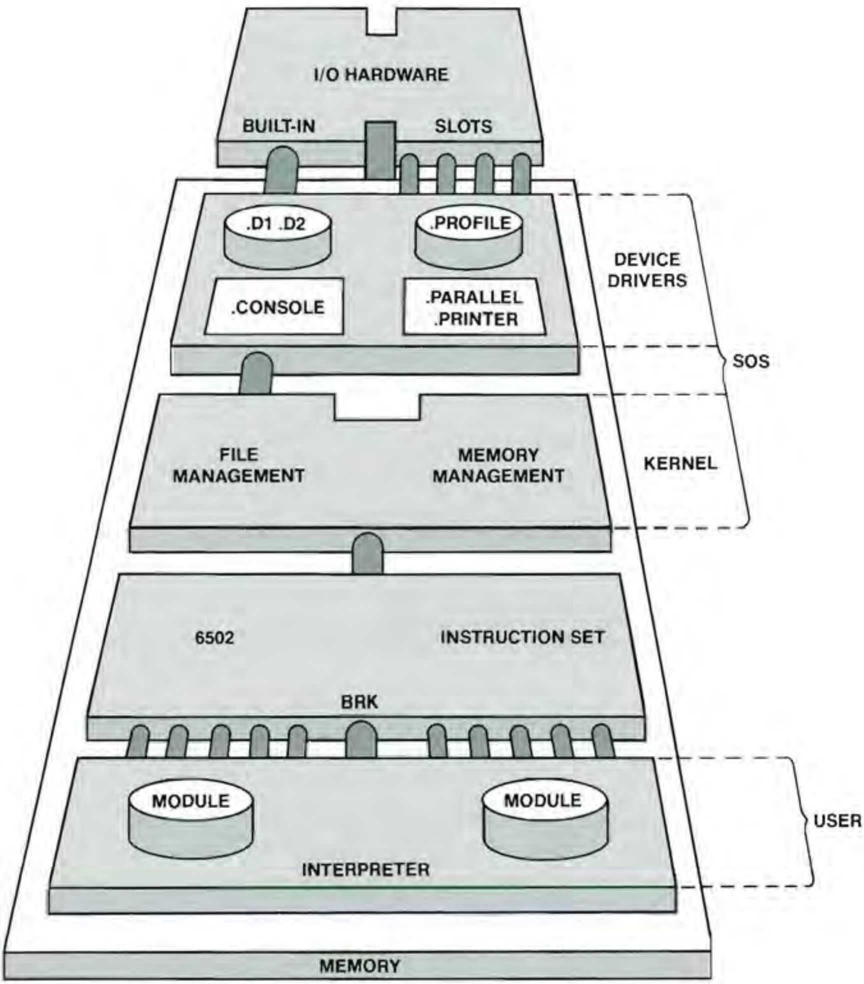


Figure 1-1. The SOS/Apple III Abstract Machine

As Figure 1-1 indicates, *almost* everything that goes on in the abstract machine does so in memory. Even the hardware attached to the abstract machine, such as printers, *appears* to exist somewhere in the machine as memory.

It is important to realize that the user's application never actually deals with any physical part of the system, it only "sees" a representation of those parts as presented to it by SOS.

SOS Data and Control Flow

Figure 1-2 shows the overall structure of SOS data and control flow. Note that all transfer of information to and from the world external to the SOS abstract machine passes through device drivers. There are no exceptions!

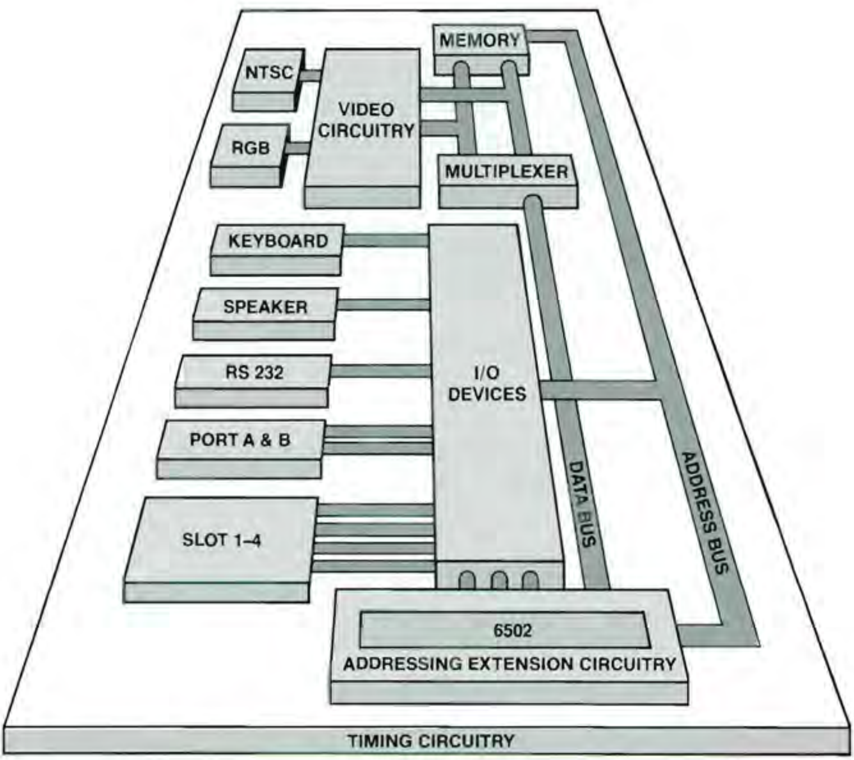


Figure 1-2. SOS Data and Control Flow

Generalized Device Driver Model

Figure 1-3 shows an idealized device driver.

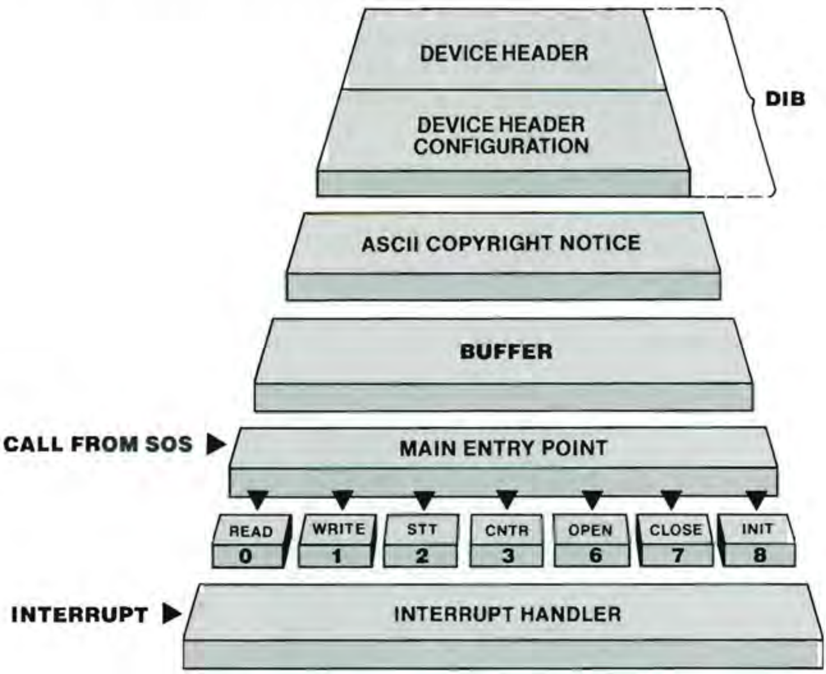


Figure 1-3. Generalized Device Driver Model

Appendices A and B in this manual contain examples of device driver skeletons that you can use as a starting point for writing your own device driver.

When you look at them, note that their structure follows that of the figure above.

Buffers (if used) must be incorporated within the body of the driver itself. When SOS places the device drivers in memory, it packs them there to maximize the use of available space. This means that a buffer outside the driver would be squeezed out by SOS.

Summary

Block device drivers support 512-byte blocks and logical block numbers. They also implement the SOS device requests DR_INIT, DR_READ, DR_WRITE, DR_STATUS, DR_CONTROL, and DR_REPEAT.

Character device drivers implement the following SOS device requests: DR_INIT, DR_OPEN, DR_CLOSE, DR_READ, DR_WRITE, DR_STATUS, and DR_CONTROL.

A device driver is part of SOS. Device drivers should be designed and tested as carefully and thoroughly as the rest of the operating system.

2

The Physical Environment of SOS

2
The Physical Environment of SOS

You should read and understand the *Apple III SOS Reference Manual* before tackling the rest of this manual.

You should be familiar with the physical environment of SOS if you are to develop efficient device drivers that can obtain the best system performance. Of particular importance in writing device drivers is familiarity with the overall memory organization and addressing of the Apple III, as well as system control registers, and how I/O devices are mapped into memory. The remainder of this chapter addresses these topics.

Hardware Diagram

Figure 2-1 is a simplified hardware diagram of the Apple III.

This figure emphasizes that the most important functional part of the Apple III is its memory. Almost everything in the system either uses or supports it.

SOS System Address Space

A portion of the diagram given in Figure 2-1 is a map of the Apple III system memory, shown in Figure 2-2.

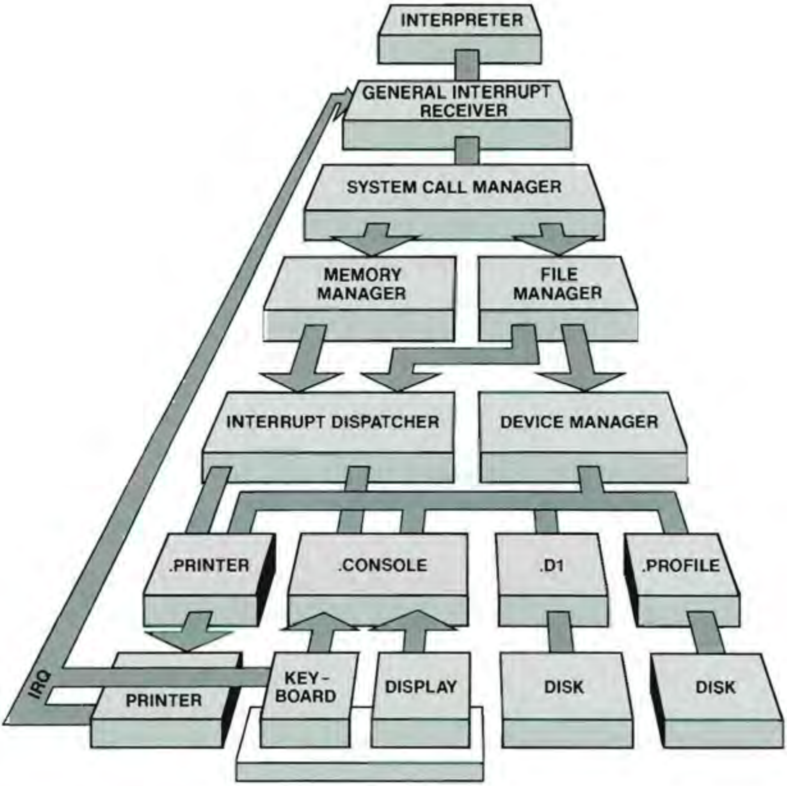


Figure 2-1. Generalized Apple III Diagram

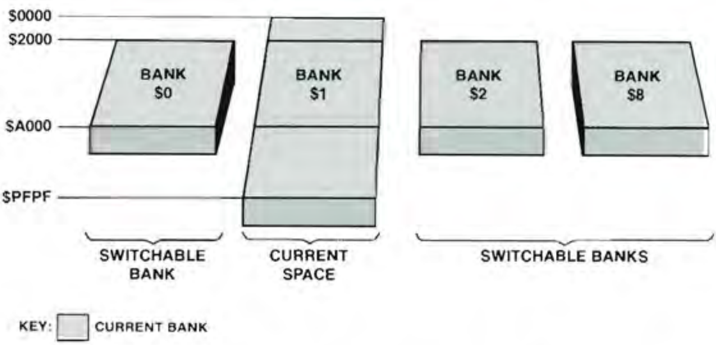


Figure 2-2. SOS System Address Space

It is important to remember that the architecture of the SOS abstract machine's memory includes these well-defined characteristics:

- One 32K block of memory, used by SOS, is always present, extending from $0000 to $1FFF and from $A000 to $FFFF.
- The remainder of memory is divided into up to 15 additional 32K blocks, each one addressed from $2000 to $9FFF. This means that the SOS abstract machine could directly address up to 512K of memory.

Note that the Apple III hardware presently supports a maximum of 256K bytes of memory.

System Control Registers

SOS has a number of registers to help it keep track of the system's state, and to aid in addressing all the memory that the system can use.

All or part of the information contained in these registers is available for your device drivers to read. The registers are described below.

E Register

The E (environment) register (at $FFDF) contains information about the state of the system. Its structure is given below, along with its usual content when a device driver is called.

Environment Register

7	6	5	4	3	2	1	0
System Clock	I/O Space	Screen State	Reset Enable	Write Protect	Stack Used	ROM Select	ROM Select

Bit	Usage	Value
7*	CPU clock rate (1 MHz or full speed)	0 (Full speed)
6	I/O space	1 (Enabled)
5	Screen	— (Undefined)
4	Reset enable	— (Undefined)
3	Write protect (top 16K)	0 (Not enabled)
2	Stack in use	1 (Primary)
1-0	ROM	00 (Deselected)

*Bit can be toggled by device drivers with reservations given below.

Because of the possible states of the screen and reset enable, the Environment register may contain values of $74, $64, $54, or $44 when a device driver is called. Your driver should change only bit 7 of the register, if necessary. The other bits should be left strictly alone.

Bit 7 defines the system clock rate, which can be switched between 1 MHz and full speed, which is presently 2 MHz.

A driver should never switch the clock to 1 MHz mode unless a part on the card that it drives is unable to handle the higher speed.

Your drivers should always reset bit 7 to zero (full speed) before exiting back to the device manager if they have had to set the clock to 1 MHz.

Z Register

The Z (zero-page) register (at $FFD0) defines the actual page in memory used for all zero-page references. It is always set to $18 when request handlers are called. When an interrupt handler is called, the Z register contains $0. See Chapter 5 for more information on interrupt handling.

This means that when you make a zero-page reference to $C0, the actual address used is $C0 of the current zero-page, an actual address of $18C0.

Enhanced-Indirect addressing requires a three-byte pointer to the desired address. The first two bytes are placed in the current zero-page while the third byte is placed in the extend-address page at the same relative address as the second byte of the address in the zero-page. The extend-address page, whose location is set by SOS, is always page $14 during driver execution.

Zero-page Register

7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0

B Register

The B (bank) register (at $FFEF) defines which of the selectable 32K banks of memory is in use by the value contained in bits 0-3. Its value is set by the system.

Since the device driver accesses memory in the bank defined by the B register, changing the register's content moves the actual area in memory being accessed to some other bank in the address space. It would be something like trying to navigate the Los Angeles freeway system while using a Chicago road map that you had just pulled out of your car's glove compartment.

Device drivers use Enhanced-Indirect addressing when passing the address of a table or list for some of the SOS driver requests (see Chapter 3).

Bank Register

7	6	5	4	3	2	1	0
	(Undefined)			(Bank in use)			

See the discussion of Enhanced-Indirect addressing later in this chapter.

Memory Addressing

The Apple III/SOS architecture allows addressing a memory space up to 512K bytes in size.

The *Apple III SOS Reference Manual* describes the Apple III addressing modes in detail. The information contained here is primarily for review of addressing modes that concern device drivers.

The two methods of addressing that concern device drivers are the Bank-switched and Enhanced-Indirect addressing modes described below.

Bank-switched Addressing

Bank-switched addressing is standard 6502 addressing except that the region of memory from $2000 through $9FFF will actually be one of up to 15 available 32K blocks of memory, depending on the value contained in the B register.

The B register always contains a value set by SOS when device drivers are called. For more information on absolute addressing, see the *Apple III Pascal Program Preparation Tools* manual.

Enhanced-Indirect Addressing

Enhanced-Indirect addressing uses a three-byte address to access any given address within the Apple III's memory, and is used by device drivers when passing pointers. It is described in detail in the *Apple III SOS Reference Manual*.

Extend-page currently in use is always equal to the content of the Z register EOR $0C. When a device driver is called, since the Z register always contains $18, the extend-page is always $14.

The first two bytes of the Enhanced-Indirect address are placed in the current zero-page ($18), and the third byte is placed in the extend-page at the same address as the high-order byte of the address in the zero-page.

The extend-byte (X-byte) may contain 0 or a value ranging from $80 to $8F, giving 16 possible values. The second half of the extend-register byte is the number of the switchable 32K bank being accessed, numbered from $0 through $F. If the extend-byte is $00, there will be no extended address in use.

After the X-byte has selected the 32K address segment to access, the two bytes in the current zero-page define the address in that segment to access. For more information on Enhanced-Indirect addressing, see the *Apple III SOS Reference Manual*.

Because of the way that extended addressing is implemented in the Apple III, locations $0000 through $00FF in any given segment cannot be addressed directly.

Here is a general algorithm for addressing those ranges of memory:

- If the address is of the form $00xx bank n, the address that you use will be of the form $80xx bank n−1.
- In the case given above, if n=0, the address that you use will be of the form $20xx bank $8F.
- If the address is of the form $FFxx bank n, the address that you use should be $7Fxx bank n+1.

An example of a program that actually implements this is given in Appendix A.

If the X-byte is $8F, the S-bank and bank 0 are switched into their normal bank-switched form. This configuration is used by graphics drivers needing to access the lowest part of the graphics area in bank 0.

RS232 Serial Port

A minimally-configured Apple III has several built-in I/O devices in addition to the keyboard and display screen. The RS232 serial port is described below.

An Asynchronous Communication Interface Adapter (ACIA) is built into the Apple III and is used for the built-in RS232 serial port. It must be accessed at the fixed 1 MHz speed.

Note that the ACIA is a 6551 and not the 6850 used in some other Apple interface devices. It contains four read/write registers that your driver can use to control the ACIA as a serial I/O device: the receive/transmit data register, status register, command register, and the control register. They are briefly described below. For more detailed information on the 6551's command, control, and status registers, see the manufacturer's data sheet.

Receive/Transmit Data Register

At $C0F0 is the receive/transmit data register. All data flowing through the Apple III's RS232 serial port passes through this register.

Status Register

The ACIA's status register is at $C0F1. It contains housekeeping information for the ACIA.

Command Register

At $C0F2 is the ACIA's command register, holding information for the ACIA on what it should be doing.

Control Register

The ACIA's control register is at $C0F3, with information on the ACIA's proper operating state.

External Device Selection

The addresses available for a given slot's I/O and onboard devices are calculated by adding the slot number multiplied by 16 to $C080. For example, slot 1 uses addresses $C090 through $C09F.

The memory addresses available to any slot (for onboard buffers, and so forth) are $Cn00 through $CnFF, where n is the number of the slot being used.

$C800 Selection

You can include up to 2K of memory decoded for the address space from $C800 on up on your interface card. Your driver can access this space by calling SELC800, which is described in Chapter 4. Since this address space may be shared among several devices, it must be explicitly allocated each time it is to be used.

The Apple III has no screen slots such as those in the Apple II available for use.

3

Request Handling

3
Request Handling

As mentioned in Chapter 1, there are two classes of device drivers: block and character. (Remember that block devices include a subclass, that of format devices.)

All device drivers handle a given set of requests passed to them by the SOS device manager through a driver request parameter table, a ten-byte list beginning at $C0 in the current zero-page.

A request handler should process the following SOS requests (assuming that its driver needs to implement them):

DR_READ
DR_WRITE
DR_STATUS
DR_CONTROL
DR_OPEN (character drivers only)
DR_CLOSE (character drivers only)
DR_INIT
DR_REPEAT (block drivers only)

After the operation has been completed, the request handler returns execution to the SOS device manager.

The request handler should also check for improper request codes, and other likely error conditions. Error handling is discussed in Chapter 4.

Device drivers are called by the SOS device manager, never by user's programs or a SOS interpreter.

Table 3-1 presents the format of the device driver parameter tables as passed to character drivers. The addresses correspond to the current zero-page in use by the device driver ($18). Note that all pointers are three-byte enhanced-indirect pointers.

DEVICE DRIVER PARAMETERS PASSED CHARACTER DRIVERS

	READ	WRITE	STATUS	CONTROL	OPEN	CLOSE	INIT
$C0	0	1	2	3	6	7	8
$C1	UNIT_NUM	UNIT_NUM	UNIT_NUM	UNIT_NUM	UNIT_NUM	UNIT_NUM	UNIT_NUM
$C2	BUFFER	BUFFER	STA CODE	CTL CODE			
$C3	POINTER	POINTER	STATUS LIST	CONTROL LIST			
$C4	REQUEST-ED	BYTE	POINTER	POINTER			
$C5	COUNT	COUNT					
$C6							
$C7							
$C8	BYTES READ						
$C9	POINTER						

NOTE: Pointers are 3-byte addresses using the X byte

Table 3-1. Character Device Driver Request Parameters

Table 3-2 presents the format of the device driver parameter tables as passed to block drivers. The addresses correspond to the current zero-page in use by the device driver ($18). Note that all pointers are three-byte enhanced-indirect pointers.

The block numbers specified in the DR_READ, DR_WRITE, and DR_REPEAT device calls are logical block numbers. Only the device driver itself knows (or cares) what the actual physical location of the data is.

DEVICE DRIVER PARAMETERS PASSED BLOCK DRIVERS

	READ	WRITE	STATUS	CONTROL		INIT	REPEAT
$C0	0	1	2	3		8	9
$C1	UNIT_NUM	UNIT_NUM	UNIT_NUM	UNIT_NUM		UNIT_NUM	UNIT_NUM
$C2	BUFFER	BUFFER	STA CODE	CTL CODE			BUFFER
$C3	POINTER	POINTER	STATUS LIST	CONTROL LIST			POINTER
$C4	REQUEST-ED	BYTE	POINTER	POINTER			
$C5	COUNT	COUNT					IGNORED
$C6	BLOCK	BLOCK					BLOCK
$C7	NUMBER	NUMBER					NUMBER
$C8	BYTES READ						
$C9	POINTER						

NOTE: Pointers are 3-byte addresses using the X byte

Table 3-2. Block Device Driver Request Parameters

The parameters passed to device drivers and their uses are further described later in this chapter in the individual descriptions of the SOS driver requests.

In addition to request handling, some drivers also handle interrupts. Interrupt handling as it relates to device drivers is described in Chapter 5 of this manual.

The first code executed in your drivers is a request handler, which is the single entry point for each device driver.

The request handler checks the contents of $C0 for the request code passed by the SOS device handler. It then branches to the appropriate part of your driver and begins acting on the request.

Driver Execution Environment

Every time a device driver is called by the device manager, some aspects of the execution environment are the same. These characteristics are outlined in Table 3-3.

The environment characteristics outlined in Table 3-3 are described in more detail below.

Zero- and Extended-address Page Usage

Zero-page locations $C0 through $FF are available for all device drivers' use. (Some of them are preloaded when your driver is called.)

Since all the drivers configured into the system share the same zero- and extend-page locations, these locations are useful to a given driver only while that driver is running. Other than the parameter list passed to the driver when it is called, your driver cannot count on the contents of the rest of the space when it begins execution.

Characteristic	State
Decimal mode	Disabled
Interrupts	Enabled
Status bits (N, V, B, Z, C)	Indeterminate
Accumulator	Indeterminate
X register	Indeterminate
Y register	Indeterminate
Environment register	
CPU clock	Full speed
I/O space	Enabled
Screen	Undefined
Reset lock	Undefined
Write protect	Off
Stack	Primary
ROM	Disabled
Zero-page in use	$18
Extend-page in use	$14
Bank register	System
I/O Expansion Slot	Deselected

Table 3-3. SOS Device Driver Environment

Driver Parameter Table

Parameters are always passed to device drivers in locations $C0 through $C9 in the current zero-page ($18). Depending on the type of driver operation being requested, all of these locations may not be used. For a complete description of each SOS driver request's parameter table, see the individual SOS driver request descriptions later in this chapter.

B Register

The B (bank) register is located at $FFEF and contains the number of the bank in which your driver resides.

System Clock State

The system clock determines how fast the Apple III operates, and its speed can be changed. It normally runs at 2 MHz (full speed), but some parts of the system cannot operate at that speed. When these parts (such as the video refresh) are working, the clock is slowed to 1 MHz.

This rapid switching between 1 and 2 MHz means that the system effectively operates somewhere between 1.4 and 1.7 MHz.

Avoid using time-dependent code! If exact timing is absolutely necessary, then hardware to take care of the critical timing functions should be on your interface card.

When your driver is called, the system clock speed is always set to full speed, and should be reset to that when you exit the driver if you have changed it. Since you cannot depend on the exact clock speed during operation in full speed mode, you can only be certain of the *minimum* time needed for any given operation to be completed.

You should never switch the clock rate to 1 Mhz unless parts of your device will not operate at higher rates.

System Interrupt State

Interrupts (IRQ) will be enabled, and unless you absolutely require them to be disabled, leave them alone. Interrupts and interrupt handlers are described in detail in Chapter 5.

System I/O State

When your driver is called, it can depend on the I/O space to be selected and $C800 space to be *not* selected.

Internal Driver Structure

All device drivers consist of a Device Information Block (DIB), storage and communication buffers (as and if needed by the driver), a request handler, an interrupt handler, and device requests.

Usual programming convention places the drivers' buffers and data before any of the executable code.

The general structure of a device driver is shown in Figure 3-1.

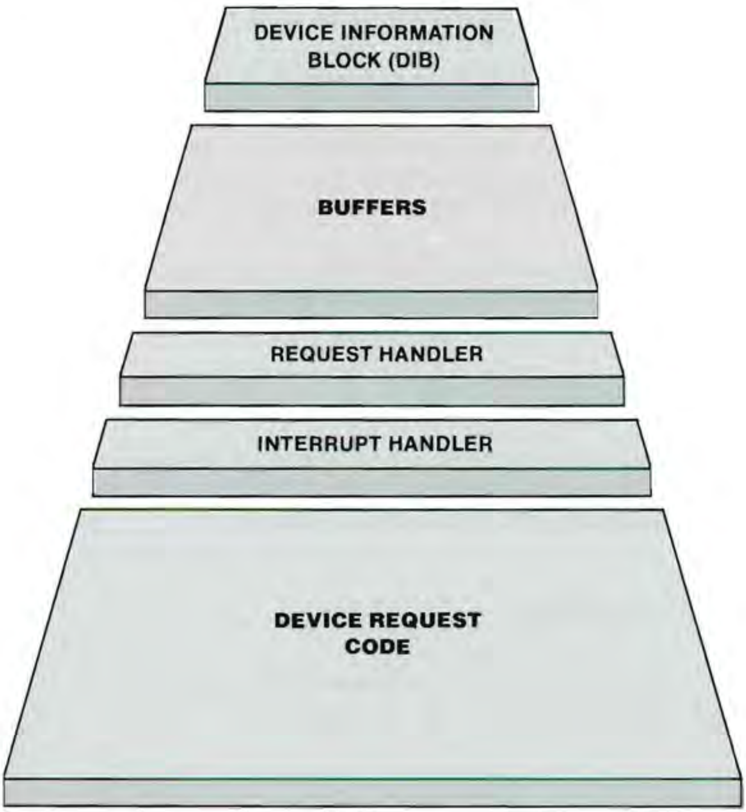


Figure 3-1. Device Driver Structure

The Device Information Block (DIB)

A DIB is a table at the beginning of each driver defining the characteristics of the devices that the driver can handle. A device driver may have more than one DIB; for example, if it handles more than one device. A DIB is made up of two parts, the header block and the configuration block, described below.

The DIB Header Block

The DIB header block is a table beginning at the first address of the driver. Table 3-4 outlines its structure.

Field Name	Length (bytes)
Comment field	3+ (optional)
Link pointer	2
Entry pointer	2
Device name (dev_name)	16
Flags	1
Slot (slot_num)	1
Unit number (unit_num)	1
Device type (dev_type)	1
Device subtype	1
Filler	1
Blocks	2
Manufacturer (manuf_id)	2
Version (ver_num)	2
Configuration field	256 (max)

Table 3-4. DIB Header Block Structure

The *Comment field* is optional. If used, it can only appear at the beginning of the the first header block in the driver. A comment field is signalled by placing $FFFF as the first two bytes of the driver. If it appears, the following byte will contain the length in bytes (up to 255) of the comment immediately following.

The *Link field* (bytes $0 and $1) points to the beginning of the next DIB contained within the device driver. If there are no more DIBs in the driver, the Link field must be set to zero. A DIB is required for each device served by a device driver.

The *Entry field* (bytes $2 and $3) points to the driver's entry address. The entry point is defined by the device driver's writer and the value is relocated during system boot to reflect the driver's location in memory after startup. This pointer is used by the SOS device manager when it calls the device driver.

The *Device name* (bytes $4 through $13) begins with a byte defining the length of the device name. The name itself is composed of a period followed by the name of the device. The first character of the name must be alphabetic, followed by any combination of alphanumeric characters and periods. Any characters in the device name field past the number defined in the count byte are ignored. All alphabetic characters must be uppercase, and no blanks are allowed in the name.

The *Flag byte* (byte $14) is examined by SOS during system startup. Bit 7 indicates whether the driver is active (1) or inactive (0), and its value can be set by SCP. Bit 6 is the Page flag and indicates whether the driver should be relocated to begin on a page boundary. Note that the byte immediately following the end of the first DIB is the one that begins the page. The other bits of the flag byte are reserved for later use and should be set to zero.

The *Slot byte* (byte $15) contains the slot number of the driver's device. (0 indicates a built-in device, such as the console). If the byte contains $FF, SCP will permit the user to modify the slot number to a value from 1 to 4, inclusive. When writing your driver, you should initialize this field to the values $00, $01 through $04, or $FF.

The *Unit byte* (byte $16) indicates the unit number of the device driver. When you write a driver, set the first DIB's unit number to 0, the second to 1, and so on.

The *Device type byte* (byte $17), along with the following byte is used for device classification and indentification. This field specifies the generic family that the device belongs to.

The device type byte for SOS character devices has the following structure:

7	6	5	4	3	2	1	0
0	W	R	0	x	x	x	x

Bit 7 is cleared for all character devices.

Bit 6 (W) is the "write allowed" byte. It must be set for all character devices that accept data from the Apple III.

Bit 5 (R) is the "read allowed" bit. It must be set for all character devices that send data to the Apple III.

Bit 4 is reserved for future use and must always be cleared.

The device type byte for SOS block devices has the following structure:

7	6	5	4	3	2	1	0
1	W	Rem	Fmt	x	x	x	x

Bit 7 is set for all block devices.

Bit 6 (W) is the "write allowed" byte. It must be set for all block devices that accept data from the Apple III.

Bit 5 (R) is the "removable device" bit. It must be set for all block devices that use removeable storage media, such as floppy-disk drives.

Bit 4 is set if the driver can also format its device.

Format devices (such as .FMTD1) are considered to be a special class of devices. Unless it would take up too much room, the format driver should be included in the device driver. The top four bits of the format device type byte are $0001. The button four bits, and the entire subtype byte must be identical to its block device.

The Device subtype byte (byte $18) indicates the specific device being referred to within the device type class specified in the previous byte. The two fields together uniquely define the device.

Device type/subtype assignments are made by the Apple Technical Support group. You should contact them if your device might fit into a type or subtype group not given in Table 3-5.

Device	Type	Subtype
Character device (write only):		
RS232 printer (.PRINTER)	$41	$01
Silentype printer (.SILENTYPE)	$41	$02
Parallel printer (.PARALLEL)	$41	$03
Sound port (.AUDIO)	$43	$01
Character device (read/write):		
System console (.CONSOLE)	$61	$01
Graphics screen (.GRAFIX)	$62	$01
Onboard RS232 (.RS232)	$63	$01
Parallel card (.PARALLEL)	$64	$01
Block devices:		
Disk III (.D1 through .D4)	$E1	$01
ProFile disk (.PROFILE)	$D1	$02
Format devices:		
Disk III (.FMTD1 ... .FMTD4)	$11	$01

Table 3-5. Currently-assigned SOS Device Types and Subtypes

The *Filler byte* (byte $19) is reserved for future use by Apple. Your driver must have this byte set to zero.

The *Blocks field* (bytes \$1A and \$1B) specifies, in hexadecimal, the number of logical blocks in a block device. This field must be set to zero if the device is a character device. If a block device can use more than one format, this field must be set either during DR_INIT or when the format to be used is known.

The *Manufacturer field* (bytes \$1C and \$1D) contains a code identifying the maufacturer of the driver. \$0000 unknown manufacturer, and \$0001–\$001F will be reserved for Apple Computer's devices. Other values are assigned by Technical Support at Apple Computer, Inc.

The *Version number field* (bytes \$1E and \$1F) contain the version number of the device driver. Its format is given below:

7	6	5	4	3	2	1	0
v1				Q			
V				v0			

In this figure V corresponds to the major version number (ranging from \$0 through \$7), v0 and v1 together correspond to the minor version number (ranging from \$0 through \$99), and Q (ranging from \$0, \$A through \$E) allows further qualification of the number. For example,

1.16C

would be represented by the following values: V=\$1, v0=\$1, v1=\$6, and Q=\$C.

The version field is followed by the DIB configuration block, described below.

The DIB Configuration Block

The DIB configuration block is an optional table following the DIB header block. It contains information about the device(s) handled by the device driver. If used, there must be a separate configuration block for each device handled by a single driver.

The first two bytes of the DIB configuration block contain the number of bytes in the block, in "low byte, high byte" order. The high byte is always $00.

The DIB configuration block content is defined by the device driver writer and can contain configuration information such as baud rate of the device, and so on. This information must be covered in the driver documentation, and its values can be altered by the System Configuration Program (SCP).

There must be a Device Configuration Block included for each physical device served by the driver if you want to be able to use SCP to alter information about the device.

Storage and Communication Buffers

You should reserve space for storage and communication buffers immediately after the DIB in your device drivers. All parts of a driver must reside in the same bank of memory. SOS packs drivers together within the bank during each system startup to most efficiently use space, and the driver's buffers must be set up within the driver itself to avoid being squeezed out of existence.

SOS Driver Requests

The major portion of a device driver is taken up by request handlers, the code that implements the SOS device requests. Each device request is implemented by a request handler.

SOS device requests are described below.

DR_INIT	Driver Request $08

DR_INIT prepares the driver's device(s) for use after system startup. It also tells SOS how many, and what type, of devices that the driver will be handling.

Parameters:

Address	Content
$C0	8
$C1	Unit number

If DR_INIT is unable to perform any of its functions, it should return to SOS with carry set. If everything is all right, DR_INIT returns with carry clear.

Note that SOS cannot handle any event queued during DR—INIT operation.

DR_OPEN	Driver Request $06

DR_OPEN is used to activate a device for use by allocating the necessary resources. It is not used by block device drivers.

Parameters:

Address	Content
$C0	6
$C1	Unit number

DR_CLOSE **Driver Request $07**

DR_CLOSE sets the specified character device to closed. It also returns the device and driver to their pre-DR_OPEN state and releases any resources that have been allocated by the driver.

DR_CLOSE is not used for block devices.

Parameters:

Address	Content
$C0	7
$C1	Unit number

The unit number is defined in the DIB header block of your device driver.

The specified unit must have been previously opened or else an error results from the call.

DR_READ **Driver Request $00**

DR_READ is used to request data from a device.

A DR_READ will take data from the device until one of the following conditions is met:

1. The requested number of bytes have been read.
2. The NEWLINE mode is active and the NEWLINE character has been encountered (this applies only to character devices).
3. The end of the data buffer has been reached.

Parameters for a character device:

Address	Content
$C0	0
$C1	Unit number
$C2–$C3	Buffer pointer
–$14C3	
$C4–$C5	Requested count
$C6–$C7	Ignored
$C8–$C9	Bytes-read pointer
–$14C9	

Parameters for a block device:

Address	Content
$C0	0
$C1	Unit number
$C2–$C3	Buffer pointer
–$14C3	
$C4–$C5	Requested count
$C6–$C7	Block number
$C8–$C9	Bytes-read pointer
–$14C9	

The buffer pointer in $C2 and $C3 refers to an area where the information being read from the device will be stored.

Locations $C6 and $C7, used only by block devices, contain the number of the logical block where the read is to begin.

The requested count ($C4–$C5) is the number of characters that are desired by the caller, and a request of 0 characters is a valid request.

$C8–$C9 points to a location containing the number of characters actually read from the device.

Note that block devices transfer data only in 512-byte blocks, and do not deal with NEWLINE mode.

DR_WRITE **Device Request $01**

DR_WRITE is used to send information to a device to be printed (or displayed, written to disk, and so forth).

Parameters for a character device:

Address	Content
$C0	1
$C1	Unit number
$C2–$C3 –$14C3	Buffer pointer
$C4–$C5	Byte count
$C6–$C7	Ignored

Parameters for a block device:

Address	Content
$C0	1
$C1	Unit number
$C2–$C3	Buffer pointer
$C4–$C5	Byte count
$C6–$C7	Block number

The buffer contains the information to be written by the device. Remember that the byte count for block devices is given in multiples of 512 bytes.

The block number (given for block devices only) is the logical number of the first block to be written.

DR_REPEAT **Driver Request $09**

DR_REPEAT is used (by block drivers only) to repeat the previous DR_READ or DR_WRITE operation.

You should include a "last request" byte somewhere in your device driver to keep track of the driver's last-performed non-DR_REPEAT operation.

Parameters:

Address	Content
$C0	9
$C1	Unit number
$C2-$C3	Buffer pointer
-$14C3	
$C4-$C5	Ignored
$C6-$C7	Block number

The block number is the logical block number at which the requested operation is to begin.

The last operation performed by that driver and the unit being called must have been either DR_READ or DR_WRITE.

DR_STATUS **Driver Request $02**

DR_STATUS is used to obtain the current status of a device or its driver.

Parameters:

Address	Content
$C0	2
$C1	Unit number
$C2	Status code
$C3-$C4	Status list pointer
-$14C4	

The content of $C2 is a status code, with different codes for character and block drivers. Character drivers must support at least the codes given below:

Status code	Meaning
$00	No operation
$01	Return control parameters
$02	Return NEWLINE information

Additional status codes may be included with a device driver, and, if added, must be described in the driver's documentation.

The structure of the status list, if used, depends on the particular status code request being performed.

For a $00 status code, the status list is a single byte:

Bit	Value	Meaning
7	0	Device not busy
	1	Device busy
6–2	—	Not used
1	0	Device (or medium) not write-protected
	1	Write-protected
0	—	Not used

For a $01 status code, the first byte of the control list contains the length of the control list in bytes. The structure and content of the remainder of the list depends on the driver. Each driver's documentation should describe its particular usage.

A $02 status code points to a two-byte list. The first byte contains $00 if there is no NEWLINE character, and $80 if there is one. The second byte in the list contains the new NEWLINE character, assuming it exists.

The control parameters returned for other status codes given below differ for each device driver. These must be included in each device driver's documentation.

Block driver status codes are:

Status code	Meaning
$00	Return status byte
$FE	Return bitmap location

For a $00 status code, the status list is a single byte:

Bit	Value	Meaning
7	0	Device not busy
	1	Device busy
6–2	—	Not used
1	0	Device (or medium) not write-protected
	1	Write-protected
0	—	Not used

For a $FE status code, the driver writes two bytes to the status list. This list will always contain $FFFF unless there is some good reason to have the volume's bitmap placed at a particular location. $FFFF means that the driver doesn't care, and the bitmap is generally placed immediately following the directory.

The length of each status list depends on the driver. It must be documented for each different driver.

DR_CONTROL **Device Request $03**

DR_CONTROL is used to send control information to a device.

Parameters:

Address	Content
\$C0	3
\$C1	Unit number
\$C2	Control code
\$C3–\$C4 –\$14C4	Control list pointer

The control code tells the device what operation it is to perform. The control list contains information that may be needed to perform the task.

The control codes passed with the DR_CONTROL call parameter list given below differ for character and block devices.

Character devices must support at least the control codes given below:

Code	Meaning
\$00	Reset device
\$01	Load control parameters
\$02	Set NEWLINE information

Control code 0 clears input and output buffers and resets the device.

Control code \$01 uses a pointer to a control list. The first byte of the list must contain the length of the list in bytes. The structure and content of a control list are peculiar to each device driver, and must be documented for each device driver.

Control code \$02 uses a two-byte control list. The first byte contains \$0 if there is no NEWLINE character, and \$80 if there is one. The second byte in the list contains the current NEWLINE character, if it exists.

For block devices, the control codes presently defined for DR_CONTROL are:

Code	Meaning
$00	Reset device
$FE	Format the device

A $00 control code is used, for example, by Pascal to perform a unit clear operation.

A $FE control code prepares the block device to read and write logical blocks of data. The position and structure of directories, if they exist, or other data structures on the device are up to the caller.

The control list must conform to the structure and content specified by the device driver being called.

4

SOS-provided Services

4

SOS-provided Services

SOS has a mechanism to handle resource contention and provide a linkage between the system's interrupt receiver and the various driver's interrupt handlers. (Interrupts and interrupt handling are described in Chapter 5 of this manual.)

A System Internal Resource (SIR) number is assigned to every function that can either generate an interrupt or must be shared among logically distinct operations handling interrupts.

Before any driver can use such a resource, it must allocate it by calling the SOS routine ALLOCSIR (described below). When the resource is no longer being used, it must be restored to the non-interrupt state and then deallocated by calling the SOS routine DEALCSIR (also described below). The present list of SIRs is given in Table 4-1.

SIR	Resource
$00	Reserved
$01	ACIA
$02–$10	Reserved
$11	Slot 1
$12	Slot 2
$13	Slot 3
$14	Slot 4

Table 4-1. System Internal Resource (SIR) Numbers

System Resource Allocation

Allocation and deallocation of system resources is provided by the SOS subroutines ALLOCSIR and DEALCSIR. Either routine may be called from any environment except an interrupt handler.

ALLOCSIR and DEALCSIR both use a table to pass the addresses of any interrupt handlers and to specify which resources are to be allocated or deallocated.

Any number of SIRs may be handled in a given call, but they should be taken in ascending numeric order. The table entry format is shown below.

Byte	Data
0	SIR number
1	ID byte
2	Interrupt handler address (high byte)
3	Interrupt handler address (low byte)
4	Interrupt handler address (X-byte)

Byte 0 of the table should contain the SIR number of the resource that you wish to be allocated or deallocated. For example, if it contains $11, the device connected to slot 1 will be allocated (or deallocated).

Byte 1 of the table contains an ID byte set by SOS that can be checked to verify ownership of the SIR. You don't need to do anything except provide space in the table for that byte.

Bytes 2 through 4 of the table contain a pointer to the beginning address of an interrupt handler for that particular resource. If there is no interrupt handler for a given SIR, the last three bytes of its entry should be zeroes.

In general, block devices are allocated during system startup, and character devices are allocated during execution of an OPEN device call by their device driver, and deallocated during execution of a CLOSE device call.

The resource-handling services provided by SOS are described below.

ALLOCSIR **Entry Point $1913**

ALLOCSIR is used to allocate System Internal Resources. The parameter table must reside in the driver's bank, and its address must specify the absolute page number.

Parameters passed:

A:	Size of parameter table in bytes
X:	Parameter table address low byte
Y:	Parameter table address high byte

Normal exit:

Carry:	Clear
A, X, Y:	Undefined

Error exit:

Carry:	Set
X:	SIR number causing error
A, Y:	Undefined

An error is caused when either the requested SIR has already been allocated or an invalid SIR is requested. If an error occurs, no SIRs are allocated.

DEALCSIR **Entry Point $1916**

DEALCSIR is used to deallocate System Internal Resources. The parameter table must reside in the driver's bank, and its address must specify the absolute page number.

Parameters passed:

A:	Size of parameter table in bytes
X:	Parameter table address low byte
Y:	Parameter table address high byte

Normal exit:

Carry:	Clear
A, X, Y:	Undefined

Error exit:

Carry:	Set
X:	SIR number causing error
A, Y:	Undefined

An error is caused when the requested SIR was not owned or an invalid SIR was requested. No SIRs are deallocated if an error occurs.

I/O Expansion Selection

The SOS subroutine SELC800 selects a peripheral card for the I/O expansion address space at $C800 through $CFFF. This subroutine may be called from any environment except an NMI interrupt handler.

The slot number of the peripheral card to be selected is passed in the accumulator and all other cards are deselected. A slot number of zero deselects all peripheral cards.

When an interrupt occurs, the SOS interrupt dispatcher automatically deselects the I/O expansion space on all peripheral cards. The previous card is reselected after the interrupt is processed. In order for this mechanism to work properly, drivers and interrupt handlers must always call SELC800 to select a peripheral card's I/O expansion space.

In addition, drivers and interrupt handlers must call SELC800 before referencing any of the I/O select addresses ($CNxx) for any peripheral card that uses the I/O expansion space.

SELC800 **Entry Point $1922**

SELC800 is used to select $C800 I/O space.

Parameters Passed:

A:	Slot number (1–4) to be selected. (0 deselects all slots.)

Normal Exit:

Carry:	Clear
A:	Undefined
X, Y:	Unchanged

Error Exit: (Invalid slot number, slot not changed.)

Carry:	Set
A, X, Y:	Unchanged

Error Handling

SOS error codes are reported by the SOS routine SYSERR. Your driver should call it whenever it encounters an error during execution. The driver will place the appropriate error code in the accumulator and then execute a JSR to SYSERR (at $1928).

SYSERR does not return to the driver after execution, but to the SOS device manager.

SYSERR **Entry Point $1928**

SYSERR is used to report errors to SOS.

Parameters Passed:

A: Error code

SYSERR does not return to the caller.

System Errors

Table 4-2 lists the presently-defined SOS error codes returned by the device driver to SOS through SYSERR.

Error Code	Meaning
$20	Invalid request code
$21	Invalid control or status code
$22	Invalid control or status parameters
$23	Device not open
$24	Device not available
$25	Resource not available
$26	Invalid operation
$27	I/O error
$28	Not connected
$2B	Write-protected
$2C	Byte count is not multiple of 512
$2D	Block number is too large
$2E	Disk switched
$30–$3F	Device-specific errors. (You define them for each device, if needed.)

Table 4-2. SOS Driver Error Codes

Event Handling

An event acts as an asynchronous interrupt in software, and drivers can define events in response to various external occurrences.

An event is armed when an interpreter requests the device driver to respond to a given condition, such as an interrupt, related to its device. The interpreter supplies the device driver with the address of a subroutine to be called when the event occurs.

When the event occurs, the driver informs SOS of the event, its priority, the address of the event handler, and then exits.

SOS then calls the event-handling routine in the interpreter.

Each time an event is signalled, an entry is made in the event queue. Then, each time the interrupt manager dispatches the user process, it checks the highest-priority entry in the event queue. If the event's priority is greater than the the user's event fence (defined in the *Apple III SOS Reference Manual*), it will be recognized and the interrupt manager will delete its entry and call the event handler.

Note that it is not presently possible to unqueue any events placed in the event queue.

When the event handler returns, the event queue is reexamined. When there are no more events above the fence, the interrupt manager restores the original user environment and returns to the user process.

Event processing is also similar to interrupt processing in that the environment is saved prior to and restored after calling the event handler, so that the user process can continue normally. The major differences are listed below:

- Events are signalled by software, interrupts by hardware.
- Event handlers are part of the user process and run in the user's environment. Interrupt handlers are part of SOS and run in SOS's interrupt environment.

- Events will only be recognized when the user process would normally be running. They never preempt SOS.
- Events are ordered. When more than one event is active at a time, they will be processed in decreasing order of priority. Events with equal priority are processed in first-in, first-out (FIFO) order.
- An event will be recognized only if its priority is greater than the current user's process event fence. The user process can raise or lower the event fence to control event recognition.

When an event is armed, the driver should save the opcode and the entry location of the event handler. When it is time to queue an event, the driver should check that location and compare its contents with the saved opcode to determine whether the event handler is still there.

Event Queueing

Events are signalled by calling the SOS subroutine QUEEVENT (described later), and may be called from any environment except an NMI interrupt handler.

When QUEEVENT is called, the event parameters are copied into an event entry, which is linked into the active event queue. Events are linked in decreasing priority, guaranteeing that the highest-priority event is always at the head of the list. The list always ends with a dummy entry with a priority of zero.

Event Recognition

SOS maintains an event fence for the user process and associates a priority with each event. Each time the event manager exits SOS and dispatches the user process, it compares the priority of the event at the head of the active event queue with the user's process current event fence. If the event's priority is greater than the event fence, the event will be recognized.

Each time control returns to SOS from an event handler, the queue is examined and succeeding events are handled until none remain in the queue above the event fence. When there are no more events to be recognized, SOS dispatches the user process.

QUEEVENT **Entry Point $191F**

The purpose of QUEEVENT is to signal an event to SOS.

Parameters passed:

X:	Parameter array address low byte
Y:	Parameter array address high byte (Must reside in current bank. If in zero-page, the high byte must specify the absolute page number, not zero.)

Normal exit (event queued):

Carry:	Clear
A, X, Y:	Undefined

The parameters passed in the parameter array are the event's priority, an ID byte (supplied by SOS) to be passed to the event handler, and the event handler's address.

The structure of the parameter array is:

Byte	Data
0	Event priority
1	ID byte (supplied by SOS)
2	Event handler address (low byte)
3	Event handler address (high byte)
4	Event handler address (X-byte)

Byte 0 contains the priority level of the event. Events with a priority level lower than the current value of the event fence are ignored.

Byte 1 is a space for an ID byte supplied by SOS to determine the ownership of any given SIR.

Bytes 2 through 4 contain a pointer to the entry point of the event handler assigned to the event in question.

5

Interrupt Handling

5
Interrupt Handling

Hardware (IRQ) interrupts allow a device driver to handle asynchronous operations in a peripheral device. By using interrupts, a device can operate more efficiently, and allow the interpreter to continue running.

For example, when you send a large number of characters to .PRINTER to be printed, the driver doesn't process all the text immediately. Instead, it immediately returns control to the interpreter, allowing the interpreter to do something else while .PRINTER processes the print buffer contents as required by the printer.

When a device interrupt occurs, SOS establishes the interrupt environment, locates the interrupt's source, and then calls the proper interrupt handler.

When the interrupt handler returns, SOS restores the saved environment and returns to the interrupted code.

Interrupt Handlers

Any device that uses or responds to interrupts requires an interrupt handler as part of its device driver.

When an interrupt handler is called, it performs three functions:

1. Clears its interrupts
2. Services the interrupting device
3. Returns to the SOS dispatcher

Interrupt Handler Design

Your interrupt handler must conform to general device driver design rules. There are some exceptions, described later, caused by slight differences in the system environment during interrupt operation.

It is up to you to make sure that the device driver and its interrupt handler operate without conflicts between each other and with SOS. Masking the interrupt when the driver is running, semaphores, or other appropriate mechanisms may be used to avoid problems, such as code reentrancy or simultaneous data access by the driver and interrupt handler.

Interrupt handlers may call only those SOS routines specifically documented as being callable from interrupt handlers.

If your interrupt handler can complete its work in about 500 microseconds or less, it should not enable the interrupt system until it has finished. However, it should never leave interrupts disabled for more than 850 microseconds. Such a case might be an indication that interrupts should not be used by the driver.

If servicing the interrupt will take more than 500 microseconds, the interrupt handler must mask its interrupt and clear the "Any Slot" interrupt flag, by storing $02 into $FFDD.

The time spent in your interrupt handler should be calculated for a clock frequency of 1 MHz. Remember that only minimum times for any process should be calculated. There is no way to guarantee *maximum* interrupt response times.

Interrupt Handler Environment

Just as during a normal call to a device driver, certain system conditions can be expected when your interrupt handler begins execution:

- Zero-page. When an interrupt occurs and your driver is called, the Z (zero-page) register will be set to $00. The extended-page used for enhanced addressing effectively does not exist during interrupt handling. Extended addressing is not available to interrupt handlers.
- Bank register. The B (bank) register ($FFEF) is set by SOS and should be left alone by your driver.
- System clock. The system clock will be set to full speed when your interrupt handler is called. After servicing the interrupt, the clock should be at full speed if your interrupt handler has changed it.
- Interrupts (IRQ). These have been disabled to allow your handler to run to completion.
- I/O space. Selected.
- I/O expansion ($C800 space). Not selected.
- Stack. The stack in use will be the primary system stack.
- X register. The processor's X register will contain a pointer to a $20-byte scratchpad area in zero-page. The scratchpad area must be addressed with ZPG,X or (ZPG,X) addressing modes.
- Y register. The processor's Y register will contain the status of the onboard ACIA that has caused the interrupt.

When two or more interrupts occur simultaneously, SOS calls the interrupt handlers in the order listed in Table 5-1.

Priority	Device
1	ACIA
2–8	Internal devices
9	Slot 1
10	Slot 2
11	Slot 3
12	Slot 4

Table 5-1. Interrupt Polling Priorities

The minimum response time to call an interrupt handler is about 160 microseconds, assuming that the interrupt system is enabled and that there are no other interrupts with a higher polling priority. When the interrupt handler returns, an additional 115 microseconds are needed to relaunch the interrupted code.

There is no guaranteed maximum response time since higher-priority interrupts may preempt lower-priority interrupts indefinitely.

Before executing, the handler should mask (or clear) its interrupt, and if the interrupt is from a peripheral slot, it must clear the "any slot" interrupt flag by storing $02 in location $FFDD.

All interrupting devices must include the ability to mask and unmask their interrupt independently of all other devices.

To prevent an interrupt handler from modifying shared data while a driver is running, the driver should mask the *device* interrupt instead of disabling the interrupt system.

In general, when you must disable the interrupt system, you should preserve the current interrupt state, disable interrupts, then restore the status. For example:

```
PHP
SEI
 :
 :
 :
PLP
```

instead of:

```
SEI
 :
 :
 :
CLI
```

Failure to follow this convention will result in unknown errors.

See the section on System Resource Allocation in Chapter 4 for more information on handling interrupts.

Interrupt Resources

SOS maintains a table of enabled IRQ interrupts and their handling routines. When a device driver become active, it can ask SOS to add an entry to this table, and give SOS the number of the interrupt it wants and the address of the interrupt handler that will respond to the interrupt.

The interrupt numbers, called SIRs, are explained in Chapter 4 under System Resource Allocation.

When SOS receives an IRQ interrupt, it polls all SIRs in order of precedence to find the particular device that generated the interrupt. It then calls the interrupt handler associated with that SIR.

An IRQ interrupt can only be enabled and serviced by a device driver.

6

Device Driver Coding Techniques

6
Device Driver Coding Techniques

Device drivers are part of SOS and they should be as reliable and as fully tested as the rest of the system.

Some things to remember when building your device drivers:

General Driver Design

When you set out to write your new driver, whether it is your first or seventy-third, there are some questions you should ask yourself.

- Is it a block or character device? This difference determines what functions it must support, how you can implement it, and how it can be tested.
- Are interrupts needed, or even useful, for your driver's operation?
- How big a buffer is needed for your device to operate most efficiently?
- What diagnostics are possible?

Device drivers hold some aspects of operation in common. All device drivers are allowed to

- Alter processor status flags D, N, V, Z, and C.
- Enable processor status I (interrupts) with some limitations as described in Chapter 5 of this manual.
- Alter A, X, and Y registers. The device manager makes no assumptions about register contents when a driver is executed.
- Alter E (environment) register except for the screen and stack bits.
- Alter the Z (zero-page) register.
- Use software loops for a guaranteed minimum timing delay.
- Disable the interrupt system by using a

```
PHP
SEI
 :
 :
 :
PLP
```

  instruction sequence.
- Absolutely must allocate slots (SIR) when their use is needed and must deallocate them when finished.

Device drivers are not allowed to

- Issue SOS calls.
- Use time-dependent code.
- Communicate with other device drivers.
- Alter the contents of the stack.
- Alter the Bank register.
- Disable the interrupt system with the sequence

```
SEI
 :
 :
 :
CLI
```

because you will lose track of the previous processor status.

Some general suggestions on designing device drivers are:

- If your driver uses interrupts (described in Chapter 5), it should mask the device interrupt to prevent the request handler and interrupt handler from conflicting over shared data.
- When you need time-dependent operations, use on-board hardware timers or a dedicated microprocessor.
- Don't depend on actual processor speed in full-speed mode. It varies.
- And finally, make things easier for yourself by using the device driver skeletons provided in Appendices A and B.

Writing Character Drivers

The list that follows gives a suggested sequence of steps for you to follow when implementing a character device driver.

- Do overall design. All character device drivers must support NEWLINE mode.
- Design tests and diagnostics.
- Begin coding.
- Implement DR_INIT.
- Start using ExerSOS to test the driver's interface with SOS. (ExerSOS is described in the *Apple III SOS Reference Manual.*)
- Implement DR_READ and DR_WRITE.
- Implement DR_STATUS and DR_CONTROL.

- Test with ExerSOS and diagnostics.
- Test with live system.

Writing Block Drivers

The list that follows gives a suggested sequence of steps for you to follow when implementing a block device driver.

- Do overall design. All block device drivers must support 512-byte blocks and logical block numbers.
- Design tests and diagnostics.
- Begin coding.
- Implement DR_INIT.
- Start using ExerSOS to test the driver's interface with SOS. (ExerSOS is described in the *Apple III SOS Reference Manual*.)
- Implement DR_READ and DR_WRITE.
- Implement DR_STATUS and DR_CONTROL.
- Implement DR_REPEAT.
- Test with ExerSOS and diagnostics.
- Test with live system.

Writing for Interrupt-driven Devices

See Chapter 5 of this manual.

Creating Device Driver Code Files

Device driver code files are produced with the Apple III Pascal Assembler. All you have to do is produce a standard relocatable object file as described in the *Apple III Pascal Program Preparation Tools manual*.

To be used as a device driver, your code file must not have been manipulated by either the Linker or the Librarian. If it has been, it will not work.

Error Detection and Reporting

It is up to your driver to catch errors during its execution.

When an error has been encountered and recognized, it must be reported to SOS through SYSERR, described in Chapter 4 under Error Handling.

Before reporting errors to SOS, which effectively terminates driver execution, you can perform any necessary housekeeping functions to insure that the driver will operate properly when it is called later on.

In addition to being able to recognize normal SOS errors, your driver must be able to recognize error conditions peculiar to the device being driven. A number of error code values have been reserved for these device-dependent errors.

The documentation describing your device driver must include a description of any special error codes for the benefit of interpreters using your device driver.

7

Interfacing with Apple III Peripheral Connectors

7

Interfacing with Apple III Peripheral Connectors

The Apple III has four peripheral connectors at the back edge of the main board that allow you to plug in peripherals to expand the usefulness of the computer. The connectors' physical and electrical characteristics are described in the following sections of this chapter.

Every peripheral card used by the Apple III requires a device driver.

Most developers of new Apple III peripherals will want to use the Apple III OEM Prototyping Card (described later in this chapter) to aid in development. All descriptions in this chapter assume that you are using the Prototyping Card for your initial development.

Physical Description

The four peripheral connectors along the back edge of the Apple III's main logic board are 50-pin PC card edge connectors with pins on 0.10" centers (Winchester 2HW25C0-111). The connector pinout appears in Figure 7-1.

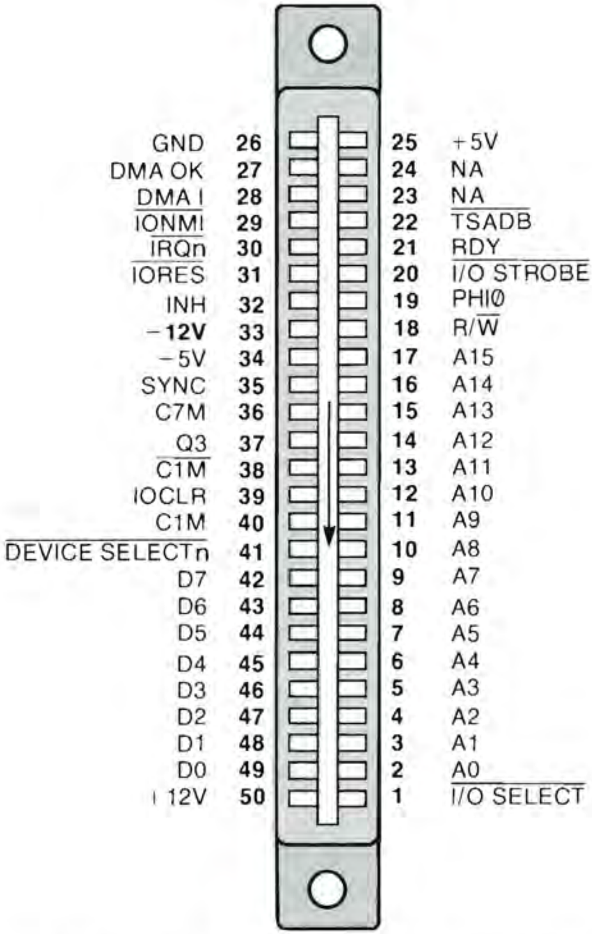


Figure 7-1. Apple III Peripheral Connector Pinout

Electrical Description

Table 7-1 specifies the signals of each pin of the Apple III peripheral connector.

Pin Number	Pin Name	In or Out**	Description
1	$\overline{\text{I/O SELECTn}}$	O	This line goes low on slot n whenever page $Cn is referenced, where n is a slot number. This signal become active during Phi0 (nominally 500 ns at 1 MHz, 250 ns during 2 MHz), and can drive a maximum of 10 LSTTL loads per peripheral card.
2–17	A0–A15	O	Buffered system address bus. Addresses are set up by the 6502 within 300 ns after the beginning of C1M. These lines can drive up to 5 LSTTL loads per peripheral card.
18	R/$\overline{\text{W}}$	I,O	READ/WRITE line. Goes high during a read cycle, and low during a write cycle. This line can drive up to 2 LSTTL loads per peripheral card.
19	PH0	O	Phi0 is a variable 1 or 2 MHz signal (depending on the current clock speed of the Apple III). The line is connected to the video timing generator's SYNC signal. It may drive up to 5 LSTTL loads per interface card.
20	$\overline{\text{I/O STROBE}}$	O	This line will go low on all peripheral connectors during Phi0 of a read or write cycle to any address in the range C800–$CFFF. This line will drive up to 4 LSTTL loads per peripheral card.
21	RDY	I	"Ready" line to the 6502. This line should change only during C1M, and when low will halt the microprocessor at the next READ cycle. This line has a 1K ohm pullup to +5V.
22	$\overline{\text{TSADB}}$	I	Any peripheral pulling this line low causes the address bus to tri-state for DMA. This line has a 1K ohm pullup to +5V.
23		NA	Not used in Apple III.
24		NA	Not used in Apple III.
25	+5V	O	Positive 5 volt supply, providing a total maximum of 600 mA. A suggested limit per card is 150 mA.
26	GND	NA	System electrical ground. (0 volt line from power supply.)

Table 7-1. Signal Description for Peripheral I/O Connectors

Pin Number	Pin Name	In or Out**	Description
27	DMAOK	O	Acknowledge signal. It informs the peripheral that the DMA requested by the peripheral can now proceed.
28	DMAI	I	Direct Memory Access (DMA) Interrupt request. This line has a 1K ohm pullup to +5V.
29	$\overline{\text{IONMI}}$	I	Input/Output Non-Maskable Interrupt. The non-maskable interrupt does not go directly to the processor, so it can be masked by the system reset lock function.
30	$\overline{\text{IRQ}}$n	I	Interrupt request line. The interrupt cycle will begin if interrupts have not been disabled. Each peripheral's signal goes to an individual gate input and can be driven by a normal TTL output.
31	$\overline{\text{IORES}}$	O	The Input/Output Reset signal is used to reset peripheral devices. It is pulled low by a power-on, Reset during Emulation mode, or a Control-Reset.
32	$\overline{\text{INH}}$	I	Inhibit line. When a device pulls this line low, all system memory is disabled. This line has a 1K ohm pullup to +5V.
33	−12V	O	Negative 12 volt supply*. The maximum current that may be drawn on this line is 150 mA.
34	−5V	O	Negative 5 volt supply*. The maximum current that may be drawn on this line is 150 mA.
35	SYNC	O	Sync is the 6502 synchronization signal. You can use it for external bus control signals.
36	C7M	O	7 MHz clock. This line will drive 2 LSTTL loads per card.
37	Q3	O	2 MHz asymmetric clock signal. This line will drive 2 LSTTL loads per peripheral card.
38	$\overline{\text{C1M}}$	O	Complement of C1M (Constant 1 MHz) clock. This line will drive up to 12 LSTTL loads per card.

Table 7-1. Signal Description for Peripheral I/O Connectors

Pin Number	Pin Name	In or Out**	Description
39	IOCLR	O	Provides the $C800 space disable function directly without address decoding. It is addressed at $C02X. ($CFFF was used as the address for disabling the expansion ROM. You should use IOCLR to ensure greater reliability for your device.)
40	C1M	O	Phase C1M (Constant 1 MHz clock). This is a constant 1 MHz at all times, regardless of system operational mode. When the system is in the 1 MHz mode, this is the same as the microprocessor Phi0 clock. This line will drive up to 12 LSTTL loads per card.
41	$\overline{\text{DEVICE SELECT}}$n	O	A read or write to addresses $C0n0 through $C0nF (where n is the slot number) causes Pin 41 on the selected connector to go low during Phi0 (400 ns in 1 MHz mode; 250 ns in 2 MHz mode).
42–49	D0–D7	I,O	Buffered bidirectional data bus. During a write cycle, data is set up by the processor 300 ns or less after the beginning of C1M. Data must be ready no less than 100 ns before the end of C1M during a read cycle.
50	+12V	O	Positive 12 volt supply, this line can supply a total maximum current of 800 mA.

*Note: Total power drawn by any one peripheral card must not exceed 1.5 watts.

**Indicates the direction of the signal: I means input to the Apple III from the peripheral; O means output from the Apple III to the peripheral; I,O means either direction is possible (for example, R/W or data).

n is the slot number on slot-specific signals.

Table 7-1. Signal Description for Peripheral I/O Connectors

Design Techniques for Interface Cards

The Apple III Prototyping card has +5V and ground (GND) available on both sides of the card. If other voltages are needed, you must wire them individually. Integrated-circuit (IC) sockets are recommended for peripheral interface applications. Transistor-Transistor Logic (TTL) should be low-power Schottky (74LS---) where possible.

Decoupling

All voltages on your card should be decoupled with a 0.1 microfarad capacitor to ground near the I/O connector card power pin at the four special locations provided. Use additional 0.1 microfarad capacitors for approximately every two low-power Schottky, CMOS, or MOS devices.

If either PROM or buffer power-down is used, the power-down circuit should be individually decoupled on the power supply side. Do *not* decouple the switched power pin.

I/O Loading and Drive Rules

Table 7-2 gives the drive and loading requirements for the peripheral I/O connector in terms of low-power Schottky logic (LSTTL). Note that MOS devices usually do not have sufficient drive for a fully loaded Apple III bus and must be buffered onto the data bus (see Table 7-2).

The address bus, the data bus, and the read/write (R/W) lines should be driven by tri-state buffers such as the 74LS365.

Pin Number	Pin Name	Drive Required By Apple III Bus	Maximum LSTTL Load*
1	$\overline{\text{I/O SELECTn}}$	N/A	12
2–17	A0–A15	Tri-State Buffer	8
18	R/$\overline{\text{W}}$	Tri-State Buffer	10
19	PH0	N/A	5
20	$\overline{\text{I/O STROBE}}$	N/A	12
21	RDY	Open Collector	N/A
22	$\overline{\text{TSADB}}$	Open Collector	N/A
23	not used	N/A	N/A
24	not used	N/A	N/A
25	+5V	N/A	N/A [150 mA]**
26	GND	N/A	N/A
27	DMAOK	N/A	4
28	DMAI	Open Collector	4
29	$\overline{\text{IONMI}}$	Open Collector	N/A
30	$\overline{\text{IRQn}}$	Open Collector	N/A
31	$\overline{\text{IORES}}$	N/A	12
32	$\overline{\text{INH}}$	Open Collector	N/A
33	−12V	N/A	N/A [50 mA]**
34	−5V	N/A	N/A [50 mA]**
35	SYNC	N/A	10
36	C7M	N/A	10
37	Q3	N/A	10
38	$\overline{\text{C1M}}$	N/A	12
39	IOCLR	N/A	12
40	C1M	N/A	12
41	$\overline{\text{DEVSELn}}$	N/A	12
42–49	D0–D7	Tri-State Buffer	8
50	+12V	N/A	N/A [75 mA]**

*Loading is per slot with reference to the main logic board. For example, each Apple III bus data line will drive 8 LSTTL inputs on any peripheral slot card.

**The power supply currents are the maximums for each card slot.

n is the slot number on slot-specific signals.

Table 7-2. Loading and Driving Rules

Since considerable capacitance is distributed over an interface card, the load contributed by up to three other peripheral cards should be considered in the design. Attempting to use PIAs and ACIAs directly on the address bus will generally lead to errors in timing and level. Type 2316 ROMs or 2716 EPROMs are exceptions, because the device timing allows them a very large margin.

Timing Signals

A number of system timing signals are available on the Apple III bus. Figure 7-2 shows details of the relative timing of these signals.

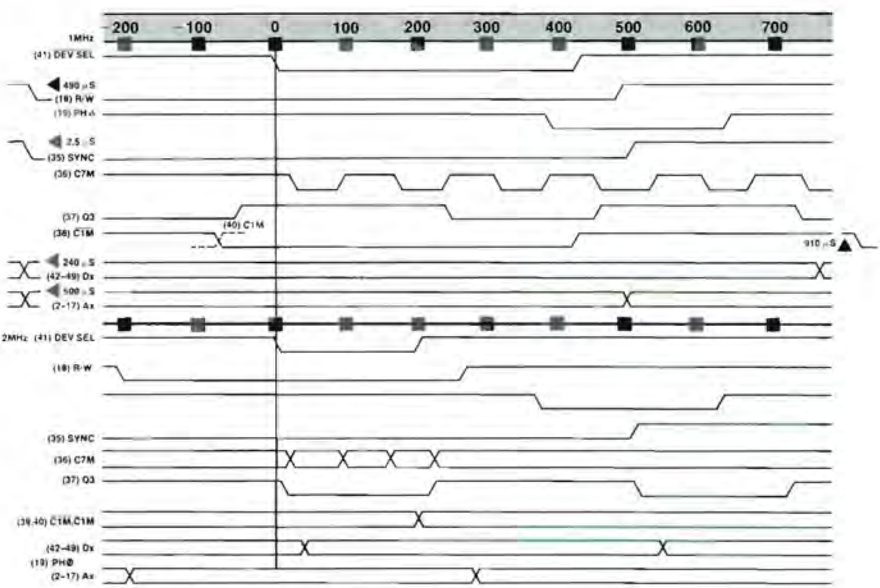


Figure 7-2. I/O Timing Diagram

The Apple III runs in two clock modes: the 1 MHz mode, and the full-speed mode, which is characterized by rapid changes of clock frequency between 1 MHz and full speed. The Apple III can be forced to operate in the 1 MHz mode either by using a special code (see Chapter 3) or by using Apple II Emulation mode. If it is in the 1 MHz mode, the Apple III strobes are about 440 nsec long and are synchronized with the 1 MHz clock.

In the normal Apple III full-speed mode, the strobes are half the length of the 1 MHz mode, as shown in Figure 7-2. More importantly, in certain applications the phase of the 1 MHz clock (pins 38 and 40) is unpredictable relative to the strobes. To perform a counting operation requiring the system 1 MHz clock to start at a precise time during a strobe, the 1 MHz mode must be used during the strobe operation.

Designing-in 6522s

The VIA LSI circuit (6522) has proven very useful for Apple-compatible peripherals. While similar to the 6520, the 6522 requires more precise timing of its clock signal.

Both circuits must be buffered to the Apple III bus for reliable operation in loaded systems. Unlike the Apple II's IRQ line, which might be "seeing" any number of LSTTL inputs, the Apple III's IRQ line sees only a single LSTTL input and thus requires no buffering.

The 6522 (and 6520) cannot be accessed in full-speed mode. Since timing margins have essentially been halved, there is insufficient time for the 6522 to latch addresses.

Figures 3 through 5 show examples of circuits using the 6522 and the 6520 that are known to work satisfactorily.

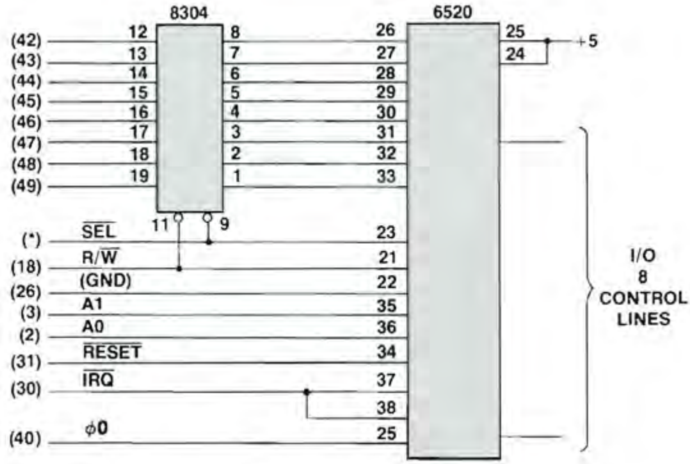


Figure 7-3. Sample 6520 Interfacing Circuit

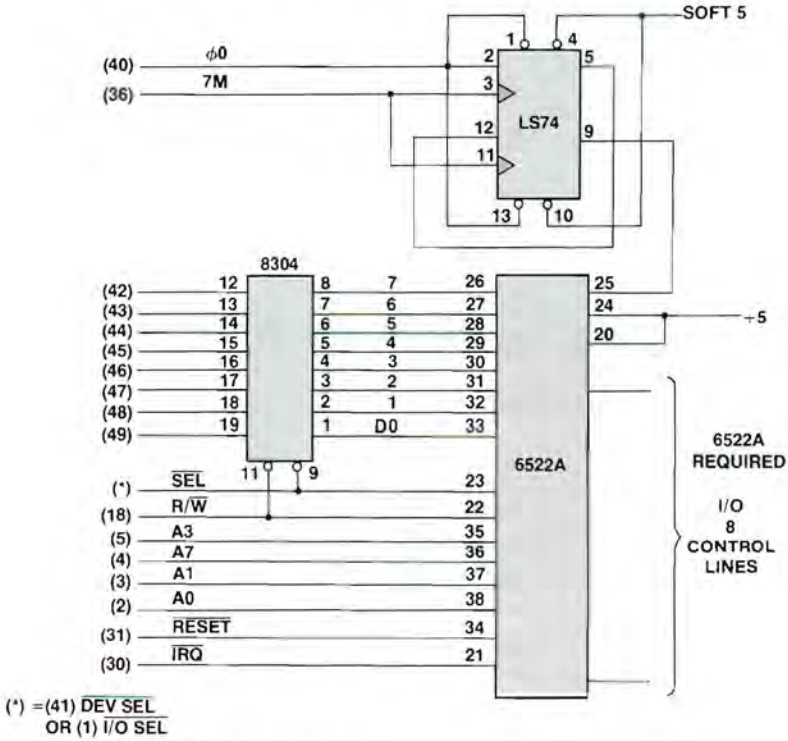


Figure 7-4. Sample (A) 6522 Interfacing Circuit

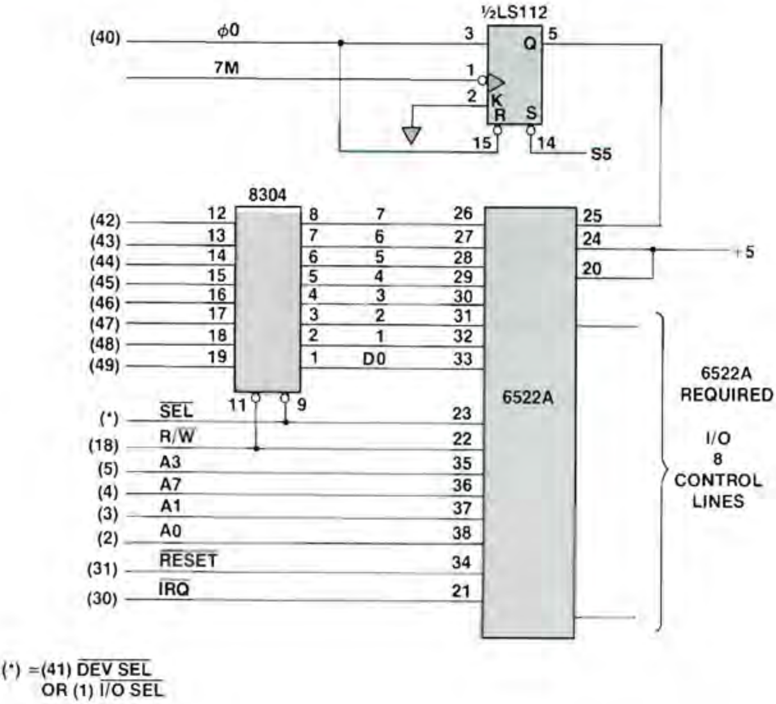


Figure 7-5. Sample (B) 6522 Interfacing Circuit

Design Techniques for Apple III Prototyping Cards

The Apple III Prototyping card is designed specifically to aid you in developing new interfaces for the Apple III. A detailed description of the card and recommended techniques for developing new interfaces is covered in the manual that is supplied with the card.

Minimizing EMI

The Apple III has been designed to minimize electromagnetic interferance (EMI) to radio and television receivers, and meets Federal Communications Commission requirements for computing devices.

Since Apple has no control over any circuitry you might design, you have to assume responsibility for "good engineering practice" and any EMI generated by the interface card.

Here are some guidelines to help you minimize EMI in your interface card designs:

1. Cards having no external I/O connections generally won't cause increases in external EMI. Even so, decoupling capacitors or networks should be placed on the card to reduce electrical noise coupling into the main logic board or adjacent interface cards.
2. If your card is used to interface an external peripheral to the Apple III, extra precautions must be taken because data signals on I/O cables are a significant source of EMI.

External I/O connections must be of the metal shell–type, such as the "DB" connector family. It is important to use metal-shell connectors on both the card and the I/O cable.

The connector on your interface card should have the metal shell electrically connected to logic ground. This may be accomplished by using I-brackets to mount the connector on the cord. The metal shell of the connector should also be electrically connected to the metal casting of the Apple III at the rear I/O port.

All I/O cables must be of the shielded type (preferably braided shield over pre-insulated signal conductors).

DO NOT use unshielded flat ribbon cables!
Due to cable construction techiniques, there is an exposed (unshielded) area between the cable shield and the connector. The cable shield must be connected to the metal shell of the connector by using short jumper wires.

Similar construction techniques should be used at the peripheral end of the cable.

Testing

Although the Apple III computer is tolerant of normal handling and use, certain conditions will lead to damage of the main logic board or its components. Before installing a new prototype interface card, it is very important to check for short circuits (or other miswiring) to prevent damage.

The test for short circuits on the constructed card has two steps:

1. Check for short circuits between the power supply lines and ground on the card by using an ohmmeter. Also check all power supply traces, whether they are used or not, before installing any ICs or transistors.
2. Check for short circuits between each I/O connector trace and all other connector traces on both sides of the board. One typical board short circuit occurs between traces that are on opposite sides of the connector.

Once you are certain that the power supply and I/O connector traces won't be short circuited, you can install the card and continue testing as follows:

1. Turn off the Apple III's power switch on the back of the computer. Unplug the Apple III's power cable. Note the Light-Emitting Diode (LED) on the main logic board near the I/O connectors. Be sure that this LED is off before inserting or removing anything.

2. Install the card in the appropriate I/O slot.

3. Reconnect the power cable, turn the power switch back on, and check to see if the system will boot. If you have tested for short circuits correctly as described above, failure to boot probably means that there is a short circuit in the bus interface or incorrect interface logic. Remove the bus and address interface logic devices and try to boot the system again.

4. If you still can't boot the system, you probably have a serious connection or logic problem. Remove all the ICs, and try to boot the system again. If the system still does not boot, then carefully recheck your logic and wiring.

5. Your device driver may have a bug that is taking the system down during DR_INIT.

Programming Notes

The requirements for successful I/O operations depend on whether the Apple III is to be used in Native mode or Apple II Emulation mode.

Because the Apple III uses memory overlays and is RAM oriented, the only areas that are guaranteed not to be overwritten are the device driver areas. Although it is generally not considered good practice to make self-modifying code, placing the buffers and parameter storage within the driver areas is the only way to guarantee their integrity under all operating conditions.

The 6502 performs a read cycle twice at indexed locations (such as \$C080 + \$n0). The first of these is a false read. Similarly, indexed store cycles will cause a false read cycle followed by the write cycle. These false reads can disturb the status register of peripheral devices such as PIAs or ACIAs. See the *6502 Programming Manual* for details on indexed memory operations.

A

Sample Block Driver Skeleton

This appendix contains a skeletal block driver to study as an example of the structure of a basic block driver.

The sample is written for the Apple III Pascal Assembler and is representative of SOS device drivers that have been written in the past.

The implementation of the individual device requests, interrupt handling, and so on, obviously is dependent on the actual device being written for.

A
Sample Block Driver Skeleton

```
Current memory available     23454
0000:                                  title  "Apple /// Skeleton BLOCK Driver"
2 blocks for procedure code  22184 words left

0000:                                  proc    BLOCKDR
Current memory available:    22929
0000:                                  nopatchlist
0000:                                  nomacrolist
0000:
0000:                         ; Apple /// Skeleton BLOCK Driver
0000:
0000:                         ; SOS Equates
0000:
0000: 1913                    AllocSIR        EQU    1913          ; allocate system internal resource
0000: 1916                    DealcSIR        EQU    1916          ; deallocate system internal resource
0000: 1922                    SelC800         EQU    1922          ; select/deselect I/O space
0000: 1928                    SysErr          EQU    1928          ; report error to system
0000: FFDF                    EREG            EQU    0FFDF         ; environment register
0000: FFEF                    BREG            EQU    0FFEF         ; bank register
0000:
0000: 00C0                    REQCODE         EQU    0C0           ; request code
0000: 00C1                    SOSUNIT         EQU    0C1           ; unit number
0000: 00C2                    SOSBUF          EQU    0C2           ; buffer pointer
0000: 00C4                    REQCNT          EQU    0C4           ; requested byte count
0000: 00C2                    CTLSTAT         EQU    0C2           ; control/status code
0000: 00C3                    CSLIST          EQU    0C3           ; control/status list pointer
0000: 00C6                    SOSBLK          EQU    0C6           ; starting block number
0000: 00C8                    BREAD           EQU    0C8           ; bytes read returned by D_READ
0000:
0000:                         ; Our temps in zero page
0000:
0000: 00D0                    BUFFER          EQU    0D0           ; my buffer ptr
0000: 00D2                    BLOCK           EQU    0D2           ; my block ptr
0000: 00D4                    NBYTES          EQU    0D4           ; # bytes to transfer for debugs
0000: 00D5                    NBLKS           EQU    0D5           ; # blocks to transfer for r/w
0000:
0000:                         ; SOS Error Codes
0000:
0000: 0020                    XREQCODE        EQU    20            ; Invalid request code
0000: 0021                    XCTLCODE        EQU    21            ; invalid control/status code
0000: 0022                    XCTLPARAM       EQU    22            ; invalid control/status param
0000: 0025                    XNORESRC        EQU    25            ; Resource not available
0000: 0026                    XBADOP          EQU    26            ; invalid operation
0000: 0027                    XIOERROR        EQU    27            ; I/O error
0000: 0028                    XNODRIVE        EQU    28            ; drive not connected
0000: 002C                    XBYTECNT        EQU    2C            ; Byte count not multiple of 512
0000: 002D                    XBLKNUM         EQU    2D            ; Block number too large

0000:                                  page
0000:
0000:                         ; Switch Macro
0000:
0000:                                  MACRO   switch
0000:                                  IF      "%1" <> ""          ; if param1 is present
0000:                                 LDA      %1                  ; load A with switch index
0000:                                  ENDC
0000:                                 CMP      #%2+1               ; do bounds check
0000:                                 BCS      $010
0000:                                 ASL      A
```

```
0000:                                TAY
0000:                                LDA      %3+1,Y                ; get switch index from table
0000:                                PHA
0000:                                LDA      %3,Y
0000:                                PHA
0000:                                 IF      "%4" <> "*"           ; if param 4 omitted
0000:                                RTS                            ; go to code
0000:                                 ENDC
0000:                         $010    ENDM
0000:
0000:                         ; Force 1 MhZ mode
0000:
0000:                                 MACRO   set1mhz
0000:                                PHP
0000:                                SEI
0000:                                LDA      EREG
0000:                                ORA      #80
0000:                                STA      EREG
0000:                                PLP
0000:                                 ENDM
0000:
0000:                         ; Force 2 MhZ mode
0000:
0000:                                 MACRO   set2mhz
0000:                                PHP
0000:                                SEI
0000:                                LDA      EREG
0000:                                AND      #7F
0000:                                STA      EREG
0000:                                PLP
0000:                                 ENDM
0000:
0000:                         ; Gross debug call
0000:
0000:                                 MACRO   imat
0000:                                PHP
0000:                                PHA
0000:                                LDA      #%1
0000:                                STA      400
0000:                                STA      SOFAR
0000:                                PLA
0000:                                PLP
0000:                                 ENDM

0000:                                 page
0000:
0000:                         ; Device Identification Block (DIB)
0000:
0000:                         ; ****
0000:                         ; *
0000:                         ; * For block devices, fill in # blocks, type/subtype, slot, version, manuf
0000:                         ; *
0000:                         ; ****
0000:
0000: 0000                    DIB           WORD    0000          ; link
0002: ****                                  WORD    Entry         ; entry point
0004: 06                                    BYTE    6             ; name count
0005: 2E 42 4C 4F 43 4B 20                  ASCII   ".BLOCK          " ; device name
000C: 20 20 20 20 20 20 20
0013: 20
0014: 80                                    BYTE    80            ; active, no page alignment
0015: FF                      DIB_SLOT      BYTE    0FF           ; slot number
0016: 00                                    BYTE    00            ; unit number
0017: D1                                    BYTE    0D1           ; type
0018: 05                                    BYTE    005           ; subtype
0019: 00                                    BYTE    00            ; filler
001A: 8002                    DIB_BLOCKS    WORD    280           ; # blocks (80*8)
001C: 0000                                  WORD    0000          ; manufacturer-unknown!
001E: 0010                                  WORD    1000          ; release-preliminary!
0020:
0020:                         ; DCB length and DCB
0020:
0020: 0100                    DCB           WORD    1             ; one byte for now
0022:
0022: 80                      DEBUG         BYTE    80            ; debugging on (80)/off (00) flag
0023:
0023:                         ; Local storage
0023:
0023: 00                      SOFAR         BYTE    00            ; gross debug
0024: 25                      INITOK        BYTE    XNORESRC      ; init went ok(00)/error code
0025: FF                      LASTOP        BYTE    0FF           ; last op for D_REPEAT calls
0026: 00                      SLOTCN        BYTE    00            ; compute CNxx and store on init
0027: 00                      SLOTCX        BYTE    00            ; compute C0X0 and store on init
0028: 0000                    DIBPTR        WORD    DIB           ; pointer to ourselves!
002A:
002A:                         ; SIR table
002A:
002A: ****                    SIRADDR       WORD    SIRTABLE
002C:
002C: 10 00 00 00 00          SIRTABLE      BYTE    10,0,0,0,0
0031: 0005                    SIRCOUNT      EQU     *-SIRTABLE
```

```
0031:                          PAGE
0031:                 ; Main entry point for the driver
0031:
0031: A5 C0          Entry   LDA     REQCODE                 ; look at request code
0033:
0033:                 ; If this is a D_INIT call (function code 8), skip the slot setup.
0033:
0033: C9 08                   CMP     #8                      ; D_INIT?
0035: F0**                    BEQ     Doit                    ; go perform D_INIT processing
0037:
0037:                 ; If debugging is enabled, put our address into (18)FD, FE, and FF
0037:
0037:                         set1mhz
0042: AD 2200                 LDA     DEBUG
0045: F0**                    BEQ     $10
0047: AD EFFF                 LDA     BREG
004A: 85 FF                   STA     0FF                     ; bank reg
004C: AD 2800                 LDA     DIBPTR
004F: 85 FD                   STA     0FD
0051: AD 2900                 LDA     DIBPTR+1
0054: 85 FE                   STA     0FE                     ; here I am!
0056:
0056:                 ; See if initialization went ok, by looking at INITOK. If it's zero, then
0056:                 ; everything went fine, otherwise it's the error code to return.
0056:
0056: AD 2400         $10     LDA     INITOK
0059: F0**                    BEQ     $60                     ; looks ok to me
005B:
005B:                 ; Return the error! Not interested in doing business with you!
005B:
005B: 20 2819         $50     JSR     SysErr                  ; not tonight, I have a headache
005E:
005E:                 ; Select our slot. NOTE: we've slowed down to 1MhZ mode already! IMPORTANT!
005E:
005E: AD 1500         $60     LDA     DIB_SLOT                ; GOT to DOWNSHIFT before looking
0061: 20 2219                 JSR     SelC800                 ; at the slot! This one, please
0064: B0F5                    BCS     $50                     ; what! I can't have it! Oops!
0066:
0066:                 ; Now call the dispatcher as a subroutine, with the slot all set up
0066:
0066: 20 ****                 JSR     Doit
0069:
0069:                 ; Remember the operation we performed for D_REPEAT processing
0069:
0069: A5 C0                   LDA     REQCODE
006B: 8D 2500                 STA     LASTOP
006E:
006E:                 ; Release the slot, go back 2MhZ mode, and leave.
006E:
006E: A9 00                   LDA     #0
0070: 20 2219                 JSR     SelC800
0073:                         set2mhz
007E: 60                      RTS                             ; Bye.


007F:                          page
007F:                 ; The Dispatcher. Does It depending on REQCODE. Note that if we came in on
007F:                 ; a D_INIT call, we do a branch to Doit; normally, Doit is called as a
007F:                 ; subroutine! We copy the buffer pointer and block # from the parameter
007F:                 ; area into our own temps, as the system seems to want them left ALONE.
007F:
007F: A5 C2           Doit    LDA     SOSBUF
0081: 85 D0                   STA     BUFFER
0083: A5 C3                   LDA     SOSBUF+1
0085: 85 D1                   STA     BUFFER+1
0087: AD C314                 LDA     SOSBUF+1401
008A: 8D D114                 STA     BUFFER+1401             ; buffer pointer is 3 bytes!
008D: A5 C6                   LDA     SOSBLK
008F: 85 D2                   STA     BLOCK
0091: A5 C7                   LDA     SOSBLK+1
0093: 85 D3                   STA     BLOCK+1                 ; block # is only 2.
0095:
0095:                         switch  REQCODE,9,DoTable       ; go do it.
00A6:
00A6: A9 20           BadReq  LDA     #XREQCODE               ; bad request code!
00A8: 20 2819                 JSR     SYSERR                  ; Pfui!
00AB:
00AB: A9 26           BadOp   LDA     #XBADOP                 ; invalid operation!
00AD: 20 2819                 JSR     SYSERR                  ; Pfui!
00B0:
00B0:                 ; Dispatch table for Doit. One entry per command number, with holes.
00B0:
00B0: ****            DoTable  WORD   DRead-1                 ; 0 read
00B2: ****                     WORD   DWrite-1                ; 1 write
00B4: ****                     WORD   DStatus-1               ; 2 status
00B6: ****                     WORD   DControl-1              ; 3 control
00B8: A500                     WORD   BadReq-1                ; 4 unused!
00BA: A500                     WORD   BadReq-1                ; 5 unused!
00BC: AA00                     WORD   BadOp-1                 ; 6 open! not for me!
00BE: AA00                     WORD   BadOp-1                 ; 7 close! not for me!
00C0: ****                     WORD   DInit-1                 ; 8 init
00C2: ****                     WORD   Drepeat-1               ; 9 repeat
00C4:
00C4:                 ; Processing D_REPEAT is easy. Repeat the last operation if it was D_READ
00C4:                 ; or a D_WRITE, else complain.
```

```
00C4:
00C4: AD 2500        DRepeat LDA     LASTOP              ; the last thing we did
00C7: F0**                   BEQ     $1                  ; 00 is a read, that's ok
00C9: C9 01                  CMP     #1                  ; 1 is a write
00CB: F0**                   BEQ     $1                  ; that's ok too
00CD: A9 26                  LDA     #XBADOP             ; else pfui!
00CF: 20 2819                JSR     SysErr              ; complain if not a write.
00D2:
00D2:                ; Last op was read or write, jam that back in and bail through Doit again!
00D2:
00D2: 85 C0          $1      STA     REQCODE             ; simple
00D4: 4C 7F00                JMP     Doit


00D7:                          page
00D7:                ; D_INIT call processing
00D7:
00D7:                ; Called at system init time only. Check DIB_SLOT to make sure that the user
00D7:                ; set a valid slot number for our interface. Allocate it by calling AllocSIR.
00D7:                ; If everything goes ok, set INITOK to 00, else leave an error code in it.
00D7:
00D7: AD 1500        Dinit   LDA     DIB_SLOT
00DA: 30**                   BMI     $1                  ; oops! negative! that's no good!
00DC: 09 C0                  ORA     #0C0
00DE: 8D 2600                STA     SLOTCN
00E1:
00E1:                ; Compute the system internal resource number (SIR) and call AllocSIR to
00E1:                ; try and grab that for us. It performs slot checking as a side effect.
00E1:
00E1: AD 1500                LDA     DIB_SLOT
00E4: 18                     CLC
00E5: 6D 2C00                ADC     SIRTABLE            ; sir=16+slot#
00E8: 8D 2C00                STA     SIRTABLE
00EB: A9 03                  LDA     #SIRCOUNT
00ED: AE 2A00                LDX     SIRADDR
00F0: AC 2B00                LDY     SIRADDR+1
00F3: 20 1319                JSR     AllocSIR            ; this one's mine!
00F6: B0**                   BCS     $2                  ; then again, maybe it isn't!
00FB:
00FB:                ; Select the slot to see if there's a card out there
00FB:
00FB:                        setimhz                     ; downshift first!
0103: AD 1500                LDA     DIB_SLOT
0106: 20 2219                JSR     SelC800             ; can we select it?
0109: B0**                   BCS     $1                  ; b/nope!that's no good!
010B:
010B:                ; Compute C0X0 for this slot and save
010B:
010B: AD 1500                LDA     DIB_SLOT
010E: 18                     CLC
010F: 2A                     ROL     A
0110: 2A                     ROL     A
0111: 2A                     ROL     A
0112: 2A                     ROL     A
0113: 69 80                  ADC     #80                 ; C080 + (slot * 16)
0115: 8D 2700                STA     SLOTCX
0118:
0118:                ; ****
0118:                ; *
0118:                ; * Insert the code to initialize your card here.
0118:                ; *
0118:                ; ****
0118:
0118:                ; Deselect it, mark everything ok, and split.
0118:
0118: A9 00                  LDA     #0
011A: 8D 2400                STA     INITOK              ; everything fine.
011D: 20 2219                JSR     SelC800             ; deselect


0120: 60                     RTS                         ; goombye
0121:
0121:                ; Bad slot or something of that ilk.
0121:
0121: A9 28          $1      LDA     #XNODRIVE
0123: D0**                   BNE     $3
0125:
0125:                ; SIR not available- somebody got the slot before we did!
0125:
0125: A9 25          $2      LDA     #XNORESRC
0127:
0127:                ; Stuff the code into INITOK and report it as an error.
0127:
0127: 8D 2400        $3      STA     INITOK              ; no, it didn't go ok
012A: 20 2819                JSR     SysErr              ; doesn't return


012D:                          PAGE
012D:                ; Random support and checking routines for the block driver
012D:
012D:                ; Check REQCNT to insure it's a multiple of 512. Return with C clear if
012D:                ; it is, return with C set if not. Leaves NBLKS containing the number of
012D:                ; blocks to transfer.
012D:
012D: 38             CKCNT   SEC                         ; assume error
```

```
012E: A5 C4                   LDA    REQCNT              ; look at lsb of bytes to do
0130: D0**                    BNE    $1                  ; no good! lsb should be 00!
0132: A5 C5                   LDA    REQCNT+1            ; look at MSB
0134: 18                      CLC
0135: 6A                      ROR    A                   ; put botom bit into C, 0 into top
0136: 85 D5                   STA    NBLKS               ; save as number of blocks
0138: 60               $1     RTS                        ; C is set from ROR to mark error.
0139:
0139:                  ; Convert block number to drive, sector pair, and track. Includes testing
0139:                  ; for valid block number. Block number comes from BLOCK in ZP. Output is
0139:                  ; in DSS and TRK. C clear on return means no error; C set means block # bad
0139:
0139: A5 D2            CVTBLK LDA    BLOCK               ; compare BLOCK with DIB_BLOCKS
013B: CD 1A00                 CMP    DIB_BLOCKS
013E: A5 D3                   LDA    BLOCK+1             ; must be CC to be valid disk address
0140: ED 1B00                 SBC    DIB_BLOCKS+1
0143: B0**                    BCS    $2                  ; br/no good! Return with C set!
0145:
0145:                  ; ****
0145:                  ; *
0145:                  ; * Insert code to translate from block # to whatever your drive needs.
0145:                  ; * Suggestion: put the resulting track/sector/etc info in locals following
0145:                  ; * the DCB so you can look at it using the debug STATUS calls.
0145:                  ; *
0145:                  ; ****
0145:
0145: 18                      CLC
0146: 60               $2     RTS
0147:
0147:                  ; ****
0147:                  ; *
0147:                  ; * ReadIt and WriteIt need to be expanded into the actual transfer routines
0147:                  ; * for D_READ and D_WRITE using BUFFER, BUFFER+1, and BUFFER+1401 as the
0147:                  ; * buffer address. Routines are called to transfer 256 bytes, and SHOULD
0147:                  ; * increment BUFFER, BUFFER+1, BUFFER+1401!
0147:                  ; *
0147:                  ; ****
0147:
0147: 60               ReadIt RTS
0148:
0148: 60               WriteIt RTS

0149:                          PAGE
0149:                  ; D_READ call processing
0149:
0149: 0149             DRead   EQU    *
0149:
0149:                  ; Validate the number of bytes to transfer and turn that into # of blocks
0149:
0149: 20 2D01                 JSR    CKCNT
014C: 90**                    BCC    $15
014E:
014E:                  ; Count not multiple of 512. Complain
014E:
014E: A9 2C                   LDA    #XBYTECNT
0150: 20 2819          $10    JSR    SysErr              ; bye.
0153:
0153:                  ; Zero # bytes read
0153:
0153: A0 00            $15    LDY    #0
0155: 98                      TYA
0156: 91 CB                   STA    (BREAD),Y           ; bytes read
0158: C8                      INY
0159: 91 CB                   STA    (BREAD),Y           ; msb of bytes read
015B:
015B:                  ; Insure the buffer address won't cause us any problems
015B:
015B: 20 ****                 JSR    FixUp               ; and fix it if it did.
015E:
015E:                  ; Convert first block number to drive/sector/track
015E:
015E: 20 3901          $1     JSR    CVTBLK
0161: 90**                    BCC    $2                  ; converted ok.
0163:
0163:                  ; Block number stinks. Complain.
0163:
0163: A9 2D                   LDA    #XBLKNUM
0165: 20 2819                 JSR    SYSERR              ; bye
0168:
0168:                  ; Test number of blocks left to transfer
0168:
0168: A5 D5            $2     LDA    NBLKS
016A: D0**                    BNE    $4
016C: 60               $3     RTS                        ; all done! bye!
016D:
016D:                  ; Transfer a block from the disk to the user
016D:
016D: 20 4701          $4     JSR    ReadIt
0170: A9 27                   LDA    #XIOERROR
0172: B0DC                    BCS    $10                 ; oops! read error!
0174:
0174:                  ; Mark another 512 bytes read.
0174:
0174: A0 01                   LDY    #1
0176: B1 CB                   LDA    (BREAD),Y
0178: 69 02                   ADC    #2
```

```
017A: 91 CB                       STA     (BREAD),Y
017C:
017C:                    ; Bump the block number
017C:
017C: E6 D2                       INC     BLOCK
017E: D0**                        BNE     $5
0180: E6 D3                       INC     BLOCK+1
0182:
0182:                    ; Decrement # of blocks to do
0182:
0182: C6 D5              $5       DEC     NBLKS
0184: F0E6                        BEQ     $3                     ; quit if that's all!
0186: D0D6                        BNE     $1                     ; else do more blocks

0188:                             PAGE
0188:                    ; D_WRITE call processing
0188:
0188: 0188               DWrite   EQU     *
0188:
0188:                    ; Validate the number of bytes to transfer and turn that into # of blocks.
0188:
0188: 20 2001                     JSR     CKCNT
018B: 90**                        BCC     $15
018D:
018D:                    ; Count not multiple of 512. Complain.
018D:
018D: A9 2C                       LDA     #XBYTECNT
018F: 20 2819            $10      JSR     SysErr                 ; bye.
0192:
0192:                    ; See if the buffer pointer will cause us any problems
0192:
0192: 20 ****            $15      JSR     FixUp                  ; and fix it if it did.
0195:
0195:                    ; Convert first block number to drive/sector/track
0195:
0195: 20 3901            $1       JSR     CVTBLK
0198: 90**                        BCC     $2                     ; converted ok
019A:
019A:                    ; Block number stinks. Complain.
019A:
019A: A9 2D                       LDA     #XBLKNUM
019C: 20 2819                     JSR     SYSERR                 ; bye.
019F:
019F:                    ; Test number of blocks left to transfer
019F:
019F: A5 D5              $2       LDA     NBLKS
01A1: D0**                        BNE     $4
01A3: 60                 $3       RTS                            ; all done! bye!
01A4:
01A4:                    ; Transfer a block from the user to the disk
01A4:
01A4: 20 4801            $4       JSR     WriteIt
01A7: A9 27                       LDA     #XIOERROR
01A9: B0E4                        BCS     $10                    ; oops! write error!
01AB:
01AB:                    ; Bump the block number
01AB:
01AB: E6 D2                       INC     BLOCK
01AD: D0**                        BNE     $5
01AF: E6 D3                       INC     BLOCK+1
01B1:
01B1:                    ; Decrement # of blocks to do
01B1:
01B1: C6 D5              $5       DEC     NBLKS
01B3: F0EE                        BEQ     $3                     ; quit if that's all!
01B5: D0DE                        BNE     $1                     ; else do more blocks.

01B7:                             PAGE
01B7:
01B7:                    ; D_STATUS call processing
01B7:
01B7:                    ; We must implement two D_STATUS calls:
01B7:                    ;        0       Return status (00 says not busy)
01B7:                    ;        FE      Return preferred bitmap location (FFFF)
01B7:                    ;
01B7:                    ; Additionally, for debugging, we implement:
01B7:                    ;        80      Read from driver space
01B7:                    ;        81      Read from C0X0 space
01B7:                    ;        82      Read from CN00 space
01B7:                    ;        83      Read from C8XX space
01B7:                    ;        84      Hang solid!
01B7:
01B7: A5 C2              DStatus LDA      CTLSTAT         ; command to issue
01B9: F0**                        BEQ     DS00            ; status 00
01BB: C9 FE                       CMP     #0FE
01BD: F0**                        BEQ     DSFE            ; status FE
01BF:
01BF:                    ; check for debugging and debugging ops.
01BF:
01BF: AD 2200                     LDA     DEBUG           ; is it enabled?
01C2: F0**                        BEQ     CSNG            ; br/nope, complain
01C4: 4C ****                     JMP     DSBx            ; go look for debug calls!
01C7:
01C7:                    ; Status code no good. Complain.
```

```
01C7:
01C7: A9 21            CSNG     LDA    #XCTLCODE        ; control/status code no good
01C9: 20 2819                   JSR    SYSERR
01CC:
01CC:                  ; Return status byte. Easy
01CC:
01CC: A0 00            DS00     LDY    #0
01CE: 98                        TYA                     ; both index and data
01CF: 91 C3                     STA    (CSLIST),Y       ; poke back to interested party
01D1: 60                        RTS
01D2:
01D2:                  ; Return preferred bitmap location. We return FFFF, we don't care.
01D2:
01D2: A0 00            DSFE     LDY    #0
01D4: A9 FF                     LDA    #0FF
01D6: 91 C3                     STA    (CSLIST),Y
01D8: C8                        INY
01D9: 91 C3                     STA    (CSLIST),Y       ; return FFFF
01DB: 60                        RTS                     ; and leave.

01DC:                            PAGE
01DC:
01DC:                  ; D_CONTROL call processing
01DC:
01DC:                  ; We must implement two D_CONTROL calls:
01DC:                  ;        0       Reset device
01DC:                  ;        FE      Perform media formatting
01DC:                  ;
01DC:                  ; For debugging, we implement a few more:
01DC:                  ;        80      Write deiver space
01DC:                  ;        81      Write C0X0 space
01DC:                  ;        82      Write CNxx space
01DC:                  ;        83      Write C8xx space
01DC:
01DC: A5 C2            DControl LDA     CTLSTAT          ; what we supposed to do?
01DE: F0**                      BEQ    $10              ; nothing? that's easy!
01E0: C9 FE                     CMP    #0FE             ; formatting?
01E2: F0**                      BEQ    $10              ; that's easy too!
01E4:
01E4:                  ; check for debugging and debugging ops.
01E4:
01E4: AD 2200                   LDA    DEBUG            ; is it enabled?
01E7: F0**                      BEQ    $4               ; if so, no more commands!
01E9:
01E9: 4C ****                   JMP    DCBx             ; go check for debugs.
01EC:
01EC:                  ; Control code no good. Complain
01EC:
01EC: 4C C701          $4       JMP    CSNG
01EF:
01EF:                  ; Execute reset or media formatting call. Very simple. We don't do anything!
01EF:
01EF: 60               $10      RTS
01F0:
01F0:                            INCLUDE  MISC

01F0:                            PAGE
01F0:
01F0:                  ; Bump is called to bump the buffer pointer by one page (256 bytes)
01F0:                  ; We dink the MSB of the buffer pointe, and fall into FixUp to see if
01F0:                  ; we generated an anomaly (and fix it up)
01F0:
01F0: E6 D1            Bump     INC    BUFFER+1                  ; bump and fall into next code
01F2:
01F2:                  ; Fix up the buffer pointer to correct for any addressing anomalies!
01F2:                  ; Since we'll call Bump after each page, we just need to do the initial
01F2:                  ; checking for two cases:
01F2:                  ;                 00XX bank N -> 80XX bank N-1
01F2:                  ;                                20XX bank 8F if N was 0 (!!!)
01F2:                  ;                 FFXX bank N -> 7FXX bank N+1
01F2:
01F2: A5 D1            FixUp    LDA    BUFFER+1                  ; look at MSB
01F4: F0**                      BEQ    $2                        ; br/that's one!
01F6: C9 FF                     CMP    #0FF                      ; is it the other one?
01F8: F0**                      BEQ    $3                        ; br/yup, fix it!
01FA: 60                        RTS                              ; an easy one!
01FB:
01FB: A9 80            $2       LDA    #80                       ; 00XX -> 80XX
01FD: 85 D1                     STA    BUFFER+1
01FF: CE D114                   DEC    BUFFER+1401               ; bank N -> bank N-1
0202: AD D114                   LDA    BUFFER+1401               ; see if it was bank 0
0205: C9 7F                     CMP    #7F                       ; (80) before the DEC.
0207: D0**                      BNE    $4                        ; br/nope, all fixed
0209: A9 20                     LDA    #20                       ; if it was, change both
020B: 85 D1                     STA    BUFFER+1                  ; msb of address and
020D: A9 8F                     LDA    #8F
020F: 8D D114                   STA    BUFFER+1401               ; bank number for bank 8F (!!!)
0212: D0**                      BNE    $4                        ; always branches
0214:
0214: 18               $3       CLC
0215: 66 D1                     ROR    BUFFER+1                  ; FFXX -> 7FXX (clever coding)
0217: EE D114                   INC    BUFFER+1401               ; bank N -> bank N+1
021A: 60               $4       RTS                              ; bye.
```

```
0218:                              PAGE
0218:
0218:                       ; D_STATUS debugging calls.  These calls transfer data from the driver and
0218:                       ; its I/O space to the user buffer.  The format of the status list for these
0218:                       ; calls is:
0218:                       ;
0218:                       ;       B0 | #bytes | disp | disp | data...      Read from driver area
0218:                       ;       B1 | #bytes | disp |  00  | data...      Read from C0Xx space
0218:                       ;       B2 | #bytes | disp |  00  | data...      Read from CNxx space
0218:                       ;       B3 | #bytes | disp | disp | data...      Read from C8xx space
0218:                       ;
0218:                       ;       #bytes - number of bytes to transfer, 00 to 255
0218:                       ;
0218:                       ; For various bizarre reasons, we choose to modify the load instruction
0218:                       ; rather than use indexing.  The range checking on the various calls depends
0218:                       ; on how much code I write to do range checking.
0218:
0218:                       ; Common code.  Set up # bytes to transfer, bump CSLIST pointer, and
0218:                       ; do the transfer.  We do it in 1MhZ mode as we may be looking at the slot.
0218:
0218: 20 ****              DSBx    JSR     DSCSET          ; do setup for debug calls
021E: 90**                        BCC     $2              ; b/went ok.
0220:
0220:                       ; DSCSET didn't like something.  The error code is in A.  let's complain!
0220:
0220: 20 2819                     JSR     SysErr          ; bye.
0223:
0223:                       ; Check the number of bytes to transfer
0223:
0223: F0**                 $2      BEQ     Scram           ; split if 00 bytes to transfer!
0225:
0225:                       ; Define the instruction to do as an abs LDA
0225:
0225: A9 AD                       LDA     #0AD
0227: 8D ****                     STA     Oak             ; not the best technique
022A:
022A:                       ; set 1MhZ mode, and do the transfer
022A:
022A:                             set1mhz
0235:
0235: 20 ****              DSloop  JSR     Oak             ; go do it.
0238: 91 C3                       STA     (CSLIST),Y      ; return data to user
023A: C8                          INY
023B: EE ****                     INC     ADDRL
023E: D0**                        BNE     $1
0240: EE ****                     INC     ADDRH
0243: C6 D4                $1      DEC     NBYTES          ; bump pointers, decrement count
0245: D0EE                        BNE     DSloop          ; loop through all bytes
0247:
0247:                             set2mhz                 ; back to full speed
0252: 60                   Scram   RTS                     ; all done.


0253:                              page
0253:
0253:                       ; Setup code for both status and control debug calls.  We validate the
0253:                       ; displacement and possibly length parameters in the control/status list
0253:                       ; and set up the address in ADDRL, ADDRH in the instruction we'll execute
0253:                       ; later on to do the transfers.
0253:
0253: A0 01                DSCSET  LDY     #1              ; index used by later code
0255: A5 C2                       LDA     CTLSTAT         ; op to perform
0257: C9 B0                       CMP     #B0             ; r/w driver space?
0259: F0**                        BEQ     DSB0            ; b/yes, set up for that
025B: C9 B1                       CMP     #B1
025D: F0**                        BEQ     DSB1            ; r/w C0Xx space
025F: C9 B2                       CMP     #B2
0261: F0**                        BEQ     DSB2            ; r/w CN00 space
0263: C9 B3                       CMP     #B3
0265: F0**                        BEQ     DSB3            ; r/w C8xx space
0267: C9 B4                       CMP     #B4
0269: F0FE                 $1      BEQ     $1              ; hang solid!
026B:
026B:                       ; Not one of ours, return error code in A with C set
026B:
026B: A9 21                $2      LDA     #XCTLCODE
026D: 38                          SEC
026E: 60                          RTS
026F:
026F:                       ; Return bad parameter error.
026F:
026F: A9 22                NOPARAM LDA     #XCTLPARAM      ; parameter is no good
0271: 38                          SEC
0272: 60                          RTS
0273:
0273: 18                   DSB0    CLC                     ; read from driver
0274: AD 2800                     LDA     DIBPTR          ; point to us
0277: 71 C3                       ADC     (CSLIST),Y      ; add in first byte
0279: 8D ****                     STA     ADDRL           ; put into instruction
027C: C8                          INY
027D: AD 2900                     LDA     DIBPTR+1
0280: 71 C3                       ADC     (CSLIST),Y      ; form hi byte
0282: 8D ****                     STA     ADDRH           ; store into instruction
0285: 4C ****                     JMP     DCFin           ; go finish up
```

```
0288:
0288: B1 C3          DSB1    LDA     (CSLIST),Y      ; pick up displacement
028A: 30E3                   BMI     NOPARAM         ; that won't do!
028C: C9 10                  CMP     #10
028E: 10DF                   BPL     NOPARAM         ; nor will that! only our slot!
0290: AA                     TAX                     ; stash for a moment
0291: AD 1500                LDA     DIB_SLOT        ; what's our slot?
0294: F0D9                   BEQ     NOPARAM         ; cute, we don't have one.
0296: 0A                     ASL     A
0297: 0A                     ASL     A
0298: 0A                     ASL     A
0299: 0A                     ASL     A               ; multiply by 16
029A: 18                     CLC

029B: 69 80                  ADC     #80             ; form X0 for the slot
029D: 71 C3                  ADC     (CSLIST),Y      ; add in displacement
029F: 8D ****                STA     ADDRL           ; store low byte into instruction
02A2: C8                     INY
02A3: B1 C3                  LDA     (CSLIST),Y      ; better be 00!
02A5: D0C8                   BNE     NOPARAM         ; only your slot!
02A7: A0 00                  LDY     #0
02A9: B1 C3                  LDA     (CSLIST),Y      ; how many bytes again?
02AB: 30C2                   BMI     NOPARAM         ; nope
02AD: C8                     INY                     ; point to displacement again
02AE: 18                     CLC
02AF: 71 C3                  ADC     (CSLIST),Y      ; must be << 10
02B1: C9 10                  CMP     #10
02B3: 10BA                   BPL     NOPARAM         ; nope. won't do at all..
02B5: 4C ****                JMP     DCFin           ; go finish up
02B8:
02B8: AD 1500        DSB2    LDA     DIB_SLOT        ; read from CN00 space
02BB: F0B2                   BEQ     NOPARAM         ; must have a slot to do it though!
02BD: 09 C0                  ORA     #0C0            ; form CN
02BF: 8D ****                STA     ADDRH           ; and hose into instruction
02C2: B1 C3                  LDA     (CSLIST),Y      ; displacement
02C4: 8D ****                STA     ADDRL           ; into instruction
02C7: C8                     INY
02C8: B1 C3                  LDA     (CSLIST),Y      ; check hi byte
02CA: D0A3                   BNE     NOPARAM         ; barf if bad
02CC: F0**                   BEQ     DCFin           ; go do cleanup processing (always branches)
02CE:
02CE: B1 C3          DSB3    LDA     (CSLIST),Y      ; low bye of displacement
02D0: 8D ****                STA     ADDRL           ; poke into instruction
02D3: C8                     INY
02D4: B1 C3                  LDA     (CSLIST),Y      ; hi byte of displacement
02D6: 3097                   BMI     NOPARAM         ; no good
02D8: C9 10                  CMP     #10             ; legal range is 0-F
02DA: 1093                   BPL     NOPARAM         ; bigger is no good!
02DC: 18                     CLC
02DD: 69 C8                  ADC     #0C8
02DF: 8D ****                STA     ADDRH           ; store into instruction
02E2:
02E2:                ; Set up the number of bytes to transfer
02E2:
02E2: A0 00          DCFin   LDY     #0              ; point back at #bytes to do
02E4: B1 C3                  LDA     (CSLIST),Y      ; get it from list
02E6: AA                     TAX
02E7: 85 D4                  STA     NBYTES          ; stash in zero page
02E9:
02E9:                ; Roll the dice. Bump CSLIST pointer by 3 and assume it won't cross into
02E9:                ; an addressing anomaly. Not guaranteed to work!
02E9:
02E9: 18                     CLC
02EA: A5 C3                  LDA     CSLIST
02EC: 69 03                  ADC     #3
02EE: 85 C3                  STA     CSLIST          ; bump lo byte by 3
02F0: A9 00                  LDA     #0
02F2: 65 C4                  ADC     CSLIST+1
02F4: 85 C4                  STA     CSLIST+1        ; maybe bump hi byte.

02F6: 18                     CLC
02F7: 8A                     TXA                     ; set z/nz on # bytes, with C clear
02F8: 60                     RTS                     ; return to caller
02F9:
02F9:                ; NOTE: The following instruction is built on the fly, to be either an absolute
02F9:                ; LDA (AD) or an absolute STA (8D). The address in the instruction is modified
02F9:                ; as we go to eliminate false strobe problems on indexed instructions.
02F9:
02F9: 00             Gak      .BYTE  00              ; Opcode goes here
02FA: 00             ADDRL    .BYTE  00              ; low byte of address
02FB: 00             ADDRH    .BYTE  00              ; hi byte of address
02FC: 60                     RTS                     ; then we return
```

```
02FD:                       PAGE
02FD:
02FD:                ; D_CONTROL debugging calls. These calls transfer data to the driver and
02FD:                ; its I/O space from the user buffer. The format of the status list for these
02FD:                ; calls is:
02FD:                ;
02FD:                ;       80 : #bytes | disp | disp | data        Write to driver area
02FD:                ;       81 : #bytes | disp |   00 : data        Write to C0Xx space
02FD:                ;       82 | #bytes | disp |   00 : data.       Write to CNxx space
02FD:                ;       83 : #bytes | disp | disp | data        Write to C8xx space
02FD:                ;
02FD:                ;       #bytes - number of bytes to transfer. 00 to 255
02FD:                ;
02FD:                ; For various bizarre reasons, we choose to modify the store instruction
02FD:                ; rather than use indexing. The range checking on the various calls depends
02FD:                ; on how much code I write to do range checking.
02FD:
02FD:                ; Common code. Set up # bytes to transfer, bump CSLIST pointer, and
02FD:                ; do the transfer. We do it in 1MhZ mode as we may be looking at the slot.
02FD:
02FD: 20 5302        DCBx    JSR     DSCSET          ; go do setup
0300: 90**                   BCC     $2
0302:
0302:                ; Setup barfed. Return error code in A
0302:
0302: 20 2819                JSR     SysErr
0305:
0305: F0**           $2      BEQ     Leave           ; and scram if it's 00!
0307:
0307:                ; Define the instruction as an abs STA (blecch!)
0307:
0307: A9 8D                  LDA     #8D
0309: 8D F902                STA     Gak             ; set up as an abs STA instruction!
030C:
030C:                ; set 1MhZ mode, and do the transfer.
030C:
030C:                        set1mhz
0317:
0317: B1 C3          DCloop  LDA     (CSLIST),Y      ; pick up user data
0319: 20 F902                JSR     Gak             ; put it away.
031C: C8                     INY
031D: EE FA02                INC     ADDRL
0320: D0**                   BNE     $1
0322: EE FB02                INC     ADDRH
0325: C6 D4          $1      DEC     NBYTES          ; bump pointers, decrement count
0327: D0EE                   BNE     DCloop          ; loop through all bytes
0329:
0329:                        set2mhz                 ; back to full speed
0334: 60             Leave   RTS                     ; all done.
0335:                        END

AB - Absolute    LB - Label      UD - Undefined   MC - Macro
RF - Ref         DF - Def        PR - Proc        FC - Func
PB - Public      PV - Private    CS - Consts

ADDRH    LB 02FB:  ADDRL    LB 02FA:  ALLOCSIR AB 1913:  BADOP    LB 00AB:  BADREQ   LB 00AA:  BLOCK    AB 0002:  BLOCKDR  MC -----
BREAD    AB 00C8:  BREG     AB FFEF:  BUFFER   AB 0000:  BUMP     LB 01F0:  CKCNT    LB 0120:  CSLIST   AB 00C3:  CSNO     LB 01C7
CTLSTAT  AB 00C2:  CVTBLK   LB 0139:  DCBX     LB 02FD:  DCB      LB 0020:  DCFIN    LB 02E2:  DCLOOP   LB 0317:  DCONTROL LB 01DC
DEALCSIR AB 191A:  DEBUG    LB 0002:  DIB      LB 0000:  DIBBLOCK LB 001A:  DIBPTR   LB 0020:  DIBSLOT  LB 0015:  DINIT    LB 0007
DOIT     LB 007F:  DOTABLE  LB 0080:  DREAD    LB 0149:  DREPEAT  LB 00C4:  DS00     LB 01C1:  DS0B     LB 0273:  DS01     LB 0288
DS02     LB 0288:  DS03     LB 02CE:  DSBX     LB 0218:  DSCSET   LB 0253:  DSFE     LB 01D2:  DSLOOP   LB 0309:  DSTATUS  LB 0187
DWRITE   LB 0188:  ENTRY    LB 0031:  EREG     AB FFDF:  FIXUP    LB 01F2:  GAK      LB 02F9:  IMAT     MC -----:  INITOK   LB 0029
LASTOP   LB 0025:  LEAVE    LB 0334:  NBLKS    AB 0005:  NBYTES   AB 00D4:  NOPARAM  LB 026F:  READIT   LB 0147:  REQCNT   AB 00C4
REQCODE  AB 00C0:  SCRAM    LB 0252:  SELC800  AB 1922:  SET1MHZ  MC -----:  SET2MHZ  MC -----:  SIRADDR  LB 002A:  SIRCOUNT AB 0005
SIRTABLE LB 002C:  SLOTCN   LB 0026:  SLOTCX   LB 0027:  SOFAR    LB 0023:  SOSBLK   AB 00CA:  SOSBUF   AB 00C2:  SOSUNIT  AB 00C1
SWITCH   MC -----:  SYSERR   AB 1920:  WRITEIT  LB 0148:  XBADOP   AB 0026:  XBLKNUM  AB 002D:  XBYTECNT AB 002C:  XCTLCODE AB 0021
XCTLPARA AB 0022:  XIOERROR AB 0027:  XNODRIVE AB 0028:  XNORESRC AB 0025:  XREQCODE AB 0020:

Current minimum space is 21196 words.

Assembly complete:        882 lines
     0   Errors flagged on this Assembly
```

B

Sample Character Driver Skeleton

This appendix contains a skeletal character driver for you to study as an example of the structure of a basic character driver.

The sample driver is written to confirm to the Apple III Pascal Assembler and is representative of SOS device drivers that have been written in the past.

Complete implementation of the individual device requests, interrupt handling, and so on, obviously is dependent on the actual device being written for.

B

Sample Character Driver Skeleton

```
Current memory available:     23454
0000|                          title  "Apple /// Skeleton CHAR Driver"
2 blocks for procedure code  22184 words left

0000|                           proc    CHAR
Current memory available:    22929
0000|                           nopatchlist
0000|                           nomacrolist
0000|
0000|                     ; Apple /// skeleton CHARACTER driver
0000|
0000|                     ; SOS Equates
0000|
0000| 1913                AllocSIR        EQU     1913      ; allocate system internal resource
0000| 1916                DealcSIR        EQU     1916      ; deallocate system internal resource
0000| 1922                SelC800         EQU     1922      ; select/deselect I/O space
0000| 1928                SysErr          EQU     1928      ; report error to system
0000| FFDF                EREG            EQU     0FFDF     ; environment register
0000| FFEF                BREG            EQU     0FFEF     ; bank register
0000|
0000| 00C0                REQCODE         EQU     0C0       ; request code
0000| 00C1                SOSUNIT         EQU     0C1       ; unit number
0000| 00C2                BUFFER          EQU     0C2       ; buffer pointer
0000| 00C4                REQCNT          EQU     0C4       ; requested byte count
0000| 00C2                CTLSTAT         EQU     0C2       ; control/status code
0000| 00C3                CSLIST          EQU     0C3       ; control/status list pointer
0000| 00C6                SOSBLK          EQU     0C6       ; starting block number
0000| 00C8                BREAD           EQU     0C8       ; bytes read returned by D_READ
0000|
0000|                     ; Our temps in zero page
0000|
0000| 00D0                NBYTES          EQU     0D0       ; # bytes to transfer for debugs
0000| 00D1                RETCNT          EQU     0D1       ; returned byte count temp
0000|
0000|                     ; SOS Error Codes
0000|
0000| 0020                XREQCODE        EQU     20        ; Invalid request code
0000| 0021                XCTLCODE        EQU     21        ; invalid control/status code
0000| 0022                XCTLPARAM       EQU     22        ; invalid control/status param
0000| 0023                XNOTOPEN        EQU     23        ; device not open
0000| 0024                XNOTAVAIL       EQU     24        ; device not available
0000| 0025                XNORESRC        EQU     25        ; Resource not available
0000| 0026                XBADOP          EQU     26        ; invalid operation
0000| 0027                XIOERROR        EQU     27        ; I/O error
0000| 0028                XNODRIVE        EQU     28        ; drive not connected
0000| 004C                XEOFERROR       EQU     4C        ; end of file error

0000|                           page
0000|
0000|                     ; Macros
0000|
0000|                           MACRO   switch
0000|                           IF      "%1" <> ""        ; if param1 is present
0000|                          LDA      %1                ; load A with switch index
0000|                           ENDC
0000|                           IF "%2" <> ""             ; if param 2 is present
0000|                          CMP      #%2+1             ; do bounds check
0000|                          BCS      $010
0000|                           ENDC
0000|                          ASL      A
0000|                          TAY
```

```
0000:                         LDA     %3+1,Y               ; get switch index from table
0000:                         PHA
0000:                         LDA     %3,Y
0000:                         PHA
0000:                          IF     "%4" <> "*"          ; if param 4 omitted,
0000:                         RTS                          ; go to code
0000:                          ENDC
0000:                 $010     ENDM
0000:
0000:                 ; Force 1 MhZ mode
0000:
0000:                          MACRO  set1mhz
0000:                         PHP
0000:                         SEI
0000:                         LDA     EREG
0000:                         ORA     #80
0000:                         STA     EREG
0000:                         PLP
0000:                          ENDM
0000:
0000:                 ; Force 2 MhZ mode
0000:
0000:                          MACRO  set2mhz
0000:                         PHP
0000:                         SEI
0000:                         LDA     EREG
0000:                         AND     #7F
0000:                         STA     EREG
0000:                         PLP
0000:                          ENDM
0000:
0000:                 ; Increment 3 byte address- includes checking for basket cases.
0000:
0000:                          MACRO  INCADR
0000:                         INC     %1
0000:                         BNE     $310
0000:                         INC     %1+1
0000:                         BNE     $310              ; bank overflow?
0000:                         SEC                       ; yup!
0000:                         ROR     %1+1

0000:                         INC     %1+1401
0000:                 $310     ENDM
0000:
0000:                 ; Increment word macro
0000:
0000:                          MACRO  INW
0000:                         INC     %1
0000:                         BNE     $210
0000:                         INC     %1+1
0000:                 $210     ENDM
0000:
0000:                 ; Gross debug call
0000:
0000:                          MACRO  imat
0000:                         PHP
0000:                         PHA
0000:                         LDA     #%1
0000:                         STA     400
0000:                         STA     SOFAR
0000:                         PLA
0000:                         PLP
0000:                          ENDM

0000:                          page
0000:
0000:                 ; Device Identification Block (DIB)
0000:
0000: 0000            DIB               WORD    0000        ; link
0002: ****                              WORD    Entry       ; entry point
0004: 05                                BYTE    5           ; name count
0005: 2E 43 48 41 52 20 20              ASCII   ".CHAR           " ; device name
000C: 20 20 20 20 20 20 20
0013: 20
0014: 80                                BYTE    80          ; active, no page alignment
0015: FF              DIB_SLOT          BYTE    0FF         ; slot number
0016: 00                                BYTE    00          ; unit number
0017: 60                                BYTE    060         ; type - character, r/w
0018: 00                                BYTE    000         ; subtype
0019: 00                                BYTE    00          ; filler
001A: 0000            DIB_BLOCKS        WORD    0000        ; # blocks -none!
001C: 0000                              WORD    0000        ; manufacturer-unknown!
001E: 0010                              WORD    1000        ; release-preliminary!
0020:
0020:                 ; DCB length and DCB
0020:
0020: 0100            DCB               WORD    1           ; one byte for now
0022:
0022: 80              DEBUG             BYTE    80          ; debugging on (80)/off (00) flag
0023:
0023:                 ; Local storage
0023:
0023: 00              SOFAR             BYTE    00          ; gross debug
```

```
0024: 25              INITOK        BYTE   XNORESRC          ; init went ok(00)/error code
0025: 00              SLOTCN        BYTE   00                ; compute CNxx and store on init
0026: 00              SLOTCX        BYTE   00                ; compute C0X0 and store on init
0027: 0000            DIBPTR        WORD   DIB               ; pointer to ourselves!
0029: 00              OPENFLG       BYTE   00                ; open/closed flag
002A:
002A: 00              NLFLAG        BYTE   00                ; NEWLINE mode flag (80/00)
002B: 00              NLCHAR        BYTE   00                ; NEWLINE character
002C:
002C:                 ; SIR table
002C:
002C: ****            SIRADDR       WORD   SIRTABLE
002E:
002E: 10 00 00 00 00  SIRTABLE      BYTE   10,0,0,0,0
0033: 0005            SIRCOUNT      EQU    *-SIRTABLE

0033:                        PAGE
0033:                 ; Main entry point for the driver
0033:
0033: A5 C0           Entry   LDA     REQCODE            ; look at request code
0035:
0035:                 ; If this is a D_INIT call (function code 8), skip the slot setup.
0035:
0035: C9 08                   CMP     #8                 ; D_INIT?
0037: F0**                    BEQ     Doit               ; go perform D_INIT processing
0039:
0039:                 ; If debugging is enabled, put our address into (18)FD, FE, and FF.
0039:
0039: AD 2200                 LDA     DEBUG
003C: F0**                    BEQ     $10
003E: AD EFFF                 LDA     BREG
0041: 85 FF                   STA     0FF                ; bank reg
0043: AD 2700                 LDA     DIBPTR
0046: 85 FD                   STA     0FD
0048: AD 2800                 LDA     DIBPTR+1
004B: 85 FE                   STA     0FE                ; here I am!
004D:
004D:                 ; See if initialization went ok, by looking at INITOK. If it's zero, then
004D:                 ; everything went fine, otherwise it's the error code to return.
004D:
004D: AD 2400         $10     LDA     INITOK
0050: F0**                    BEQ     $60                ; looks ok to me
0052:
0052:                 ; Return the error! Not interested in doing business with you!
0052:
0052: 20 2819         $50     JSR     SysErr             ; not tonight, I have a headache
0055:
0055:                 ; Now call the dispatcher as a subroutine
0055:
0055: 20 ****         $60     JSR     Doit
0058: 60                      RTS                        ; Bye.

0059:                         page
0059:                 ; The Dispatcher. Does It depending on REQCODE. Note that if we came in on
0059:                 ; a D_INIT call, we do a branch to Doit; normally, Doit is called as a
0059:                 ; subroutine!
0059:
0059: 0059            Doit     EQU    *
0059:
0059:                         switch  REQCODE,8,DoTable    ; go do it.
006A:
006A: A9 20           BadReq  LDA     #XREQCODE          ; bad request code!
006C: 20 2819                 JSR     SysErr             ; Pfui!
006F:
006F: A9 26           BadOp   LDA     #XBADOP            ; invalid operation!
0071: 20 2819                 JSR     SysErr             ; Pfui!
0074:
0074: A9 23           NotOpen LDA     #XNOTOPEN          ; device not open for business!
0076: 20 2819                 JSR     SysErr
0079:
0079:                 ; Dispatch table for Doit. One entry per command number, with holes
0079:
0079: ****            DoTable  WORD   DRead-1            ; 0 read
007B: ****                     WORD   DWrite-1           ; 1 write
007D: ****                     WORD   DStatus-1          ; 2 status
007F: ****                     WORD   DControl-1         ; 3 control
0081: 6900                     WORD   BadReq-1           ; 4 unused!
0083: 6900                     WORD   BadReq-1           ; 5 unused!
0085: ****                     WORD   DOpen-1            ; 6 open
0087: ****                     WORD   DClose-1           ; 7 close
0089: ****                     WORD   DInit-1            ; 8 init

008B:                         page
008B:                 ; D_INIT call processing
008B:
008B:                 ; Called at system init time only. Check DIB_SLOT to make sure that the user
008B:                 ; set a valid slot number for our interface. Allocate it by calling AllocSIR.
008B:                 ; If everything goes ok, set INITOK to 00, else leave an error code in it.
008B:
008B: AD 1500         DInit   LDA     DIB_SLOT
008E: 30**                    BMI     $1                 ; oops! negative! that's no good!
0090: 09 C0                   ORA     #0C0
```

```
0092: 8D 2500               STA     SLOTCN
0095:
0095:               ; Select the slot to see if there's a card out there
0095:
0095:                       setlmh;                         ; downshift first!
00A0: AD 1500               LDA     DIB_SLOT
00A3: 20 2219               JSR     SelC800                 ; can we select it?
00A6: B0**                  BCS     $1                      ; b/nope'that's no good!
00A8:
00A8:               ; Compute C0X0 for this slot and save
00A8:
00A8: AD 1500               LDA     DIB_SLOT
00AB: 18                    CLC
00AC: 2A                    ROL     A
00AD: 2A                    ROL     A
00AE: 2A                    ROL     A
00AF: 2A                    ROL     A
00B0: 69 80                 ADC     #80                     ; C080 + (slot * 16)
00B2: 8D 2600               STA     SLOTCX
00B5:
00B5:               ; Deselect it, mark everything ok, and split.
00B5:
00B5: A9 00                 LDA     #0
00B7: 8D 2400               STA     INITOK                  ; everything fine
00BA: 20 2219               JSR     SelC800                 ; deselect
00BD: 60                    RTS                             ; goombye
00BE:
00BE:               ; Bad slot or something of that ilk.
00BE:
00BE: A9 28         $1      LDA     #XNODRIVE
00C0: D0**                  BNE     $3
00C2:
00C2:               ; SIR not available- somebody got the slot before we did!
00C2:
00C2: A9 25         $2      LDA     #XNORESRC
00C4:
00C4:               ; Stuff the code into INITOK and report it as an error
00C4:
00C4: 8D 2400       $3      STA     INITOK                  ; no, it didn't go ok.
00C7: 20 2819               JSR     SysErr                  ; doesn't return

00CA:                        PAGE
00CA:
00CA:               ; D_OPEN call processing
00CA:
00CA:               ; We allocate our resource at OPEN time, reset the device, and set up for
00CA:               ; data transfers
00CA:
00CA: AD 2900       DOpen   LDA     OPENFLG                 ; are we open already?
00CD: F0**                  BEQ     $1                      ; b/nope
00CF:
00CF:               ; If we're already open, complain!
00CF:
00CF: A9 24                 LDA     #XNOTAVAIL              ; not available.
00D1: 20 2819               JSR     SysErr
00D4:
00D4:               ; Compute the system internal resource number (SIR) and call AllocSIR to
00D4:               ; try and grab that for us. It performs slot checking as a side effect.
00D4:
00D4: AD 1500       $1      LDA     DIB_SLOT
00D7: 18                    CLC
00D8: 69 10                 ADC     #10                     ; sir=16+slot#
00DA: 8D 2E00               STA     SIRTABLE
00DD: A9 05                 LDA     #SIRCOUNT
00DF: AE 2C00               LDX     SIRADDR
00E2: AC 2D00               LDY     SIRADDR+1
00E5:
00E5:               ; ****
00E5:               ; *
00E5:               ; * Note: if an interrupt handler is used, the bank number must be loaded
00E5:               ; * from BREG and put into SIRTABLE. See writeup on AllocSIR.
00E5:               ; *
00E5:               ; ****
00E5:
00E5: 20 1319               JSR     AllocSIR                ; this one's mine!
00E8: B0**                  BCS     $2                      ; then again, maybe it isn't!
00EA:
00EA:               ; ****
00EA:               ; *
00EA:               ; * Insert device setup code here. If your device generates interrupts,
00EA:               ; * do it carefully!
00EA:               ; *
00EA:               ; ****
00EA:
00EA:               ; Mark we're open, and leave.
00EA:
00EA: A9 80                 LDA     #80
00EC: 8D 2900               STA     OPENFLG
00EF: 60                    RTS
00F0:
00F0:               ; Not available!
00F0:
00F0: A9 25         $2      LDA     #XNORESRC
00F2: 20 2819               JSR     SysErr
```

```
00F5:                                PAGE
00F5:
00F5:                        ; DClose processing
00F5:
00F5:                        ; Clean up everything. Wait for all writes to complete. Deallocate the
00F5:                        ; resources and go away.
00F5:
00F5: AD 2900                DClose  LDA     OPENFLG                 ; are we open?
00F8: D0**                           BNE     $1                      ; hope so!
00FA:
00FA: 4C 7400                        JMP     NotOpen                 ; gripe if we're not!
00FD:
00FD:                        ; After running down any active I/O and disabling interrupts, free the slot
00FD:
00FD:                        $1
00FD:
00FD:                        ; ****
00FD:                        ; *
00FD:                        ; * Insert rundown and termination code here. If the device generates
00FD:                        ; * interrupts, these must be disabled and cleared before DealcSIR is called
00FD:                        ; *
00FD:                        ; ****
00FD:
00FD: A9 05                          LDA     #SIRCOUNT
00FF: AE 2C00                        LDX     SIRADDR
0102: AC 2D00                        LDY     SIRADDR+1
0105: 20 1619                        JSR     DealcSIR                ; free the resource
0108: A9 00                          LDA     #0
010A: 8D 2900                        STA     OPENFLG                 ; mark us CLOSED
010D: 60                             RTS                             ; goombye

010E:                                PAGE
010E:                        ; D_READ call processing
010E:
010E: AD 2900                DRead   LDA     OPENFLG
0111: D0**                           BNE     $1                      ; b/we're open
0113: 4C 7400                        JMP     NotOpen                 ; and gripe if we're not!
0116:
0116:                        ; Zero # bytes read
0116:
0116: A9 00                  $1      LDA     #0
0118: 85 D1                          STA     RETCNT
011A: 85 D2                          STA     RETCNT+1                ; our z page temp
011C:
011C:                        ; Insure the buffer address won't cause us any problems
011C:
011C: 20 ****                        JSR     FixUp                   ; and fix it if it did.
011F:
011F:                        ; Compliment the requested byte count to make life easier.
011F:
011F: A9 FF                          LDA     #0FF                    ; form one's compliment
0121: 45 C4                          EOR     REQCNT
0123: 85 C4                          STA     REQCNT                  ; as it's easier to increment
0125: A9 FF                          LDA     #0FF                    ; and test for zero
0127: 45 C5                          EOR     REQCNT+1
0129: 85 C5                          STA     REQCNT+1
012B:
012B:                        ; The read loop. See if we terminate on requested byte count first.
012B:
012B: E6 C4                  Rloop   INC     REQCNT                  ; bump it
012D: D0**                           BNE     $1                      ; didn't go to zero.
012F: E6 C5                          INC     REQCNT+1                ; bump hi byte
0131: F0**                           BEQ     Rdend                   ; terminate on byte count!
0133:
0133:                        ; Get a byte from the device, put it in the user's buffer, increment
0133:                        ; the buffer pointer and the number of bytes returned.
0133:
0133: 20 ****                $1      JSR     GetByte
0136: A0 00                          LDY     #0
0138: 91 C2                          STA     (BUFFER),Y              ; store into user buffer
013A: 48                             PHA                             ; save byte on the stack
013B:                                INCADR  BUFFER                  ; bump the pointer
0149:                                INW     RETCNT                  ; bump return count
014F:
014F:                        ; Check for NEWLINE mode, and termination on NEWLINE character
014F:
014F: 68                             PLA                             ; chr back again
0150: 2C 2A00                        BIT     NLFLAG                  ; is newline mode set?
0153: 10D6                           BPL     Rloop                   ; br/nope,do it some more.
0155: CD 2B00                        CMP     NLCHAR                  ; if so, is this the one?
0158: D0D1                           BNE     Rloop                   ; br/nope, keep going.
015A:
015A:                        ; Terminate the read, either on byte count or newline. Move the #
015A:                        ; of returned bytes to the user, then split
015A:

015A: A0 00                  Rdend   LDY     #0
015C: A5 D1                          LDA     RETCNT                  ; lsb of returned byte count
015E: 91 C8                          STA     (BREAD),Y
0160: C8                             INY
0161: A5 D2                          LDA     RETCNT+1
0163: 91 C8                          STA     (BREAD),Y               ; return it
0165: 60                             RTS                             ; and leave.
```

```
0166:
0166:                    ; ****
0166:                    ; *
0166:                    ; * GetByte actually does the dirty work of getting a byte from the device
0166:                    ; * To be determined by the user! Note it is called in 2MhZ mode, and the
0166:                    ; * device/slot has NOT been selected.
0166:                    ; *
0166:                    ; ****
0166:
0166: 60                 GetByte RTS

0167:                            PAGE
0167:                    ; D_WRITE call processing
0167:
0167: AD 2900            DWrite  LDA     OPENFLG
016A: D0**                       BNE     $1                  ; b/we're open
016C: 4C 7400                    JMP     NotOpen             ; and gripe if we're not!
016F:
016F:                    ; See if the buffer pointer will cause us any problems
016F:
016F: 20 ****            $1      JSR     FixUp               ; and fix it if it did
0172:
0172:                    ; Compliment the requested byte count to make life easier.
0172:
0172: A9 FF                      LDA     #0FF                ; form one's compliment
0174: 45 C4                      EOR     REQCNT
0176: 85 C4                      STA     REQCNT              ; as it's easier to increment
0178: A9 FF                      LDA     #0FF                ; and test for zero
017A: 45 C5                      EOR     REQCNT+1
017C: 85 C5                      STA     REQCNT+1
017E:
017E:                    ; The write loop. See if we terminate on byte count
017E:
017E: E6 C4              Wloop   INC     REQCNT
0180: D0**                       BNE     $1                  ; br/nope.
0182: E6 C5                      INC     REQCNT+1
0184: D0**                       BNE     $1                  ; br/nope, more to write
0186:
0186:                    ; All done. Bye!
0186:
0186: 60                         RTS
0187:
0187:                    ; Get a byte from the user buffer, write it, and bump the pointer.
0187:
0187: A0 00              $1      LDY     #0
0189: B1 C2                      LDA     (BUFFER),Y          ; get byte
018B: 20 ****                    JSR     PutByte             ; get rid of it
018E:                            INCADR  BUFFER
019C:
019C:                    ; Go back and do it until the byte count goes to 00!
019C:
019C: 4C 7E01                    JMP     Wloop
019F:
019F:                    ; ****
019F:                    ; *
019F:                    ; * PutByte actually does the dirty work. Called in 2MhZ mode, with
019F:                    ; * slot/device NOT selected!
019F:                    ; *
019F:                    ; ****
019F:
019F: 60                 PutByte RTS

01A0:                            PAGE
01A0:
01A0:                    ; D_STATUS call processing
01A0:
01A0:                    ; We must implement three D_STATUS calls:
01A0:                    ;       0       No operation
01A0:                    ;       1       Return device control parameters
01A0:                    ;       2       Return NEWLINE flag and character
01A0:                    ;
01A0:                    ; Additionally, for debugging, we implement:
01A0:                    ;       80      Read from driver space
01A0:                    ;       81      Read from C0X0 space
01A0:                    ;       82      Read from CN00 space
01A0:                    ;       83      Read from C8XX space
01A0:                    ;       84      Hang solid!
01A0:
01A0: A5 C2              DStatus LDA     CTLSTAT         ; command to issue
01A2: F0**                       BEQ     DS00            ; status 00
01A4: C9 01                      CMP     #1
01A6: F0**                       BEQ     DS01            ; return device control params
01A8: C9 02                      CMP     #2
01AA: F0**                       BEQ     DS02            ; return NEWLINE flag and character
01AC:
01AC:                    ; check for debugging and debugging ops
01AC:
01AC: AD 2200                    LDA     DEBUG           ; is it enabled?
01AF: F0**                       BEQ     CSNG            ; br/nope, gripe
01B1: 4C ****                    JMP     DS8x            ; go look for debug calls!
01B4:
01B4:                    ; Status code no good. Complain.
01B4:
```

```
01B4: A9 21          CSNG     LDA     #XCTLCODE          ; control/status code no good
01B6: 20 2B19                 JSR     SYSERR
01B9:
01B9:                ; Doing nothing is easy.
01B9:
01B9: 60             DS00     RTS
01BA:
01BA:                ; Return device control parameters. To be determined by the device.
01BA:
01BA: 60             DS01     RTS
01BB:
01BB:                ; Return NEWLINE flag and character.
01BB:
01BB: A0 00          DS02     LDY     #0
01BD: AD 2A00                 LDA     NLFLAG             ; newline active/inactive flag
01C0: 91 C3                   STA     (CSLIST),Y         ; return to user
01C2: C8                      INY
01C3: AD 2B00                 LDA     NLCHAR             ; newline character
01C6: 91 C3                   STA     (CSLIST),Y         ; return that
01C8: 60                      RTS                        ; and split.

01C9:                          PAGE
01C9:
01C9:                ; D_CONTROL call processing
01C9:
01C9:                ; We must implement three D_CONTROL calls:
01C9:                ;        0       Reset device
01C9:                ;        1       Set control parameters
01C9:                ;        2       Set NEWLINE flag and character
01C9:                ;
01C9:                ; For debugging, we implement a few more:
01C9:                ;        B0      Write deiver space
01C9:                ;        B1      Write C0X0 space
01C9:                ;        B2      Write CNxx space
01C9:                ;        B3      Write C8xx space
01C9:
01C9: A5 C2          DControl LDA      CTLSTAT           ; what we supposed to do?
01CB: F0**                    BEQ     DC00               ; device reset
01CD: C9 01                   CMP     #1
01CF: F0**                    BEQ     DC01               ; set control params
01D1: C9 02                   CMP     #2
01D3: F0**                    BEQ     DC02               ; set NEWLINE flag and chr
01D5:
01D5:                ; check for debugging and debugging ops
01D5:
01D5: AD 2200                 LDA     DEBUG              ; is it enabled?
01D8: F0**                    BEQ     $4                 ; if so, no more commands!
01DA:
01DA: 4C ****                 JMP     DCB*               ; go check for debugs.
01DD:
01DD:                ; Control code no good. Complain.
01DD:
01DD: 4C B401        $4       JMP     CSNG
01E0:
01E0:                ; Set NEWLINE flag and character
01E0:
01E0: A0 00          DC02     LDY     #0
01E2: B1 C3                   LDA     (CSLIST),Y         ; the flag
01E4: 8D 2A00                 STA     NLFLAG             ; updated
01E7: C8                      INY
01E8: B1 C3                   LDA     (CSLIST),Y         ; newline character
01EA: 8D 2B00                 STA     NLCHAR
01ED: 60                      RTS                        ; easy to do
01EE:
01EE:                ; Reset the device. To be defined by the device.
01EE:
01EE: 60             DC00     RTS
01EF:
01EF:                ; Load control parameters. Defined by the device
01EF:
01EF: 60             DC01     RTS
01F0:
01F0:                          INCLUDE  MISC

01F0:                          .PAGE
01F0:
01F0:                ; Bump is called to bump the buffer pointer by one page (256 bytes).
01F0:                ; We dink the MSB of the buffer pointe, and fall into FixUp to see if
01F0:                ; we generated an anomaly (and fix it up).
01F0:
01F0: E6 C3          Bump     INC     BUFFER+1                   ; bump and fall into next code
01F2:
01F2:                ; Fix up the buffer pointer to correct for any addressing anomalies!
01F2:                ; Since we'll call Bump after each page, we just need to do the initial
01F2:                ; checking for two cases:
01F2:                ;                00XX bank N -> 80XX bank N-1
01F2:                ;                               20XX bank 8F if N was 0 (!!!)
01F2:                ;                FFXX bank N -> 7FXX bank N+1
01F2:
01F2: A5 C3          FixUp    LDA     BUFFER+1                   ; look at MSB
01F4: F0**                    BEQ     $2                         ; br/that's one!
01F6: C9 FF                   CMP     #0FF                       ; is it the other one?
01F8: F0**                    BEQ     $3                         ; br/yup, fix it!
```

```
01FA: 60                    RTS                          ; an easy one!
01FB:
01FB: A9 80          $2     LDA     #80                  ; 00XX -> 80XX
01FD: 85 C3                 STA     BUFFER+1
01FF: CE C314               DEC     BUFFER+1401          ; bank N -> bank N-1
0202: AD C314               LDA     BUFFER+1401          ; see if it was bank 0
0205: C9 7F                 CMP     #7F                  ; (80) before the DEC.
0207: D0**                  BNE     $4                   ; br/nope, all fixed.
0209: A9 20                 LDA     #20                  ; if it was, change both
020B: 85 C3                 STA     BUFFER+1             ; msb of address and
020D: A9 8F                 LDA     #8F
020F: 8D C314               STA     BUFFER+1401          ; bank number for bank 8F (!!!)
0212: D0**                  BNE     $4                   ; always branches
0214:
0214: 18             $3     CLC
0215: 66 C3                 ROR     BUFFER+1             ; FFXX -> 7FXX (clever coding)
0217: EE C314               INC     BUFFER+1401          ; bank N -> bank N+1
021A: 60             $4     RTS                          ; bye.

021B:                       /PAGE
021B:
021B:                ; D_STATUS debugging calls. These calls transfer data from the driver and
021B:                ; its I/O space to the user buffer. The format of the status list for these
021B:                ; calls is:
021B:                ;
021B:                ;      80 : #bytes : disp : disp : data        Read from driver area
021B:                ;      81 : #bytes : disp :   00 : data        Read from C0Xx space
021B:                ;      82 : #bytes : disp :   00 : data        Read from CNxx space
021B:                ;      83 : #bytes : disp : disp : data        Read from C8xx space
021B:                ;
021B:                ;      #bytes - number of bytes to transfer, 00 to 255
021B:                ;
021B:                ; For various bizarre reasons, we choose to modify the load instruction
021B:                ; rather than use indexing. The range checking on the various calls depends
021B:                ; on how much code I write to do range checking.
021B:
021B:                ; Common code. Set up # bytes to transfer, bump CSLIST pointer, and
021B:                ; do the transfer. We do it in 1MhZ mode as we may be looking at the slot
021B:
021B: 20 ****        DS8x   JSR     DSCSET               ; do setup for debug calls
021E: 90**                  BCC     $2                   ; b/went ok
0220:
0220:                ; DSCSET didn't like something. The error code is in A, let's complain!
0220:
0220: 20 2819               JSR     SysErr               bye
0223:
0223:                ; Check the number of bytes to transfer
0223:
0223: F0**           $2     BEQ     Scram                ; split if 00 bytes to transfer!
0225:
0225:                ; Define the instruction to do as an abs LDA
0225:
0225: A9 AD                 LDA     #0AD
0227: 8D ****               STA     Gak                  ; not the best technique
022A:
022A:                ; set 1MhZ mode, and do the transfer
022A:
022A:                       set1mhz
0235:
0235: 20 ****        DSloop JSR     Gak                  ; go do it
0238: 91 C3                 STA     (CSLIST),Y           ; return data to user
023A: C8                    INY
023B: EE ****               INC     ADDRL
023E: D0**                  BNE     $1
0240: EE ****               INC     ADDRH
0243: C6 D0          $1     DEC     NBYTES               ; bump pointers, decrement count
0245: D0EE                  BNE     DSloop               ; loop through all bytes
0247:
0247:                       set2mhz                      ; back to full speed
0252: 60             Scram  RTS                          ; all done

0253:                       /page
0253:
0253:                ; Setup code for both status and control debug calls. We validate the
0253:                ; displacement and possibly length parameters in the control/status list
0253:                ; and set up the address in ADDRL, ADDRH in the instruction we'll execute
0253:                ; later on to do the transfers
0253:
0253: A0 01          DSCSET LDY     #1                   ; index used by later code
0255: A5 C2                 LDA     CTLSTAT              ; op to perform
0257: C9 80                 CMP     #80                  ; r/w driver space?
0259: F0**                  BEQ     DS80                 ; b/yes, set up for that
025B: C9 81                 CMP     #81
025D: F0**                  BEQ     DS81                 ; r/w C0Xx space
025F: C9 82                 CMP     #82
0261: F0**                  BEQ     DS82                 ; r/w CN00 space
0263: C9 83                 CMP     #83
0265: F0**                  BEQ     DS83                 ; r/w C8xx space
0267: C9 84                 CMP     #84
0269: F0FE           $1     BEQ     $1                   ; hang solid!
026B:
026B:                ; Not one of ours, return error code in A with C set
026B:
```

```
026B: A9 21         $2      LDA   #XCTLCODE
026D: 38                    SEC
026E: 60                    RTS
026F:
026F:               ; Return bad parameter error
026F:
026F: A9 22         NGPARAM LDA   #XCTLPARAM     ; parameter is no good
0271: 38                    SEC
0272: 60                    RTS
0273:
0273: 18            DS80    CLC                  ; read from driver
0274: AD 2700               LDA   DIBPTR         ; point to us
0277: 71 C3                 ADC   (CSLIST),Y     ; add in first byte
0279: 8D ****               STA   ADDRL          ; put into instruction
027C: C8                    INY
027D: AD 2800               LDA   DIBPTR+1
0280: 71 C3                 ADC   (CSLIST),Y     ; form hi byte
0282: 8D ****               STA   ADDRH          ; store into instruction
0285: 4C ****               JMP   DCfin          ; go finish up
0288:
0288: B1 C3         DS81    LDA   (CSLIST),Y     ; pick up displacement
028A: 30E3                  BMI   NGPARAM        ; that won't do!
028C: C9 10                 CMP   #10
028E: 10DF                  BPL   NGPARAM        ; nor will that! only our slot
0290: AA                    TAX                  ; stash for a moment
0291: AD 1500               LDA   DIB_SLOT       ; what's our slot?
0294: F0D9                  BEQ   NGPARAM        ; cute, we don't have one
0296: 0A                    ASL   A
0297: 0A                    ASL   A
0298: 0A                    ASL   A
0299: 0A                    ASL   A              ; multiply by 16
029A: 18                    CLC

029B: 69 80                 ADC   #80            ; form X0 for the slot
029D: 71 C3                 ADC   (CSLIST),Y     ; add in displacement
029F: 8D ****               STA   ADDRL          ; store low byte into instruction
02A2: C8                    INY
02A3: B1 C3                 LDA   (CSLIST),Y     ; better be 00!
02A5: D0C8                  BNE   NGPARAM        ; only your slot!
02A7: A0 00                 LDY   #0
02A9: B1 C3                 LDA   (CSLIST),Y     ; how many bytes again?
02AB: 30C2                  BMI   NGPARAM        ; nope
02AD: C8                    INY                  ; point to displacement again
02AE: 18                    CLC
02AF: 71 C3                 ADC   (CSLIST),Y     ; must be <= 10
02B1: C9 10                 CMP   #10
02B3: 10BA                  BPL   NGPARAM        ; nope, won't do at all
02B5: 4C ****               JMP   DCfin          ; go finish up
02B8:
02B8: AD 1500       DS82    LDA   DIB_SLOT       ; read from CN00 space
02BB: F0B2                  BEQ   NGPARAM        ; must have a slot to do it though!
02BD: 09 C0                 ORA   #0C0           ; form CN
02BF: 8D ****               STA   ADDRH          ; and hose into instruction
02C2: B1 C3                 LDA   (CSLIST),Y     ; displacement
02C4: 8D ****               STA   ADDRL          ; into instruction (YUK!)
02C7: C8                    INY
02C8: B1 C3                 LDA   (CSLIST),Y     ; check hi byte
02CA: D0A3                  BNE   NGPARAM        ; barf if bad
02CC: F0**                  BEQ   DCfin          ; go do cleanup processing (always branches)
02CE:
02CE: B1 C3         DS83    LDA   (CSLIST),Y     ; low bye of displacement
02D0: 8D ****               STA   ADDRL          ; poke into instruction
02D3: C8                    INY
02D4: B1 C3                 LDA   (CSLIST),Y     ; hi byte of displacement
02D6: 3097                  BMI   NGPARAM        ; no good
02D8: C9 10                 CMP   #10            ; legal range is 0-F
02DA: 1093                  BPL   NGPARAM        ; bigger is no good!
02DC: 18                    CLC
02DD: 69 C8                 ADC   #0C8
02DF: 8D ****               STA   ADDRH          ; store into instruction
02E2:
02E2:               ; Set up the number of bytes to transfer
02E2:
02E2: A0 00         DCfin   LDY   #0             ; point back at #bytes to do
02E4: B1 C3                 LDA   (CSLIST),Y     ; get it from list
02E6: AA                    TAX
02E7: 85 D0                 STA   NBYTES         ; stash in zero page
02E9:
02E9:               ; Roll the dice. Bump CSLIST pointer by 3 and assume it won't cross into
02E9:               ; an addressing anomaly. Not guaranteed to work!
02E9:
02E9: 18                    CLC
02EA: A5 C3                 LDA   CSLIST
02EC: 69 03                 ADC   #3
02EE: 85 C3                 STA   CSLIST         ; bump lo byte by 3
02F0: A9 00                 LDA   #0
02F2: 65 C4                 ADC   CSLIST+1
02F4: 85 C4                 STA   CSLIST+1       ; maybe bump hi byte
```

```
02F6: 18                        CLC
02F7: 8A                        TXA                          ; set z/nz on # bytes, with C clear
02F8: 60                        RTS                          ; return to caller
02F9:
02F9:                 ; NOTE  The following instruction is built on the fly, to be either an absolute
02F9:                 ; LDA (AD) or an absolute STA (8D)  The address in the instruction is modified
02F9:                 ; as we go to eliminate false strobe problems on indexed instructions
02F9:
02F9: 00              Gak       BYTE    00                   ; Opcode goes here
02FA: 00              ADDRL     BYTE    00                   ; low byte of address
02FB: 00              ADDRH     BYTE    00                   ; hi byte of address
02FC: 60                        RTS                          ; then we return (Gak'!)

02FD:                           PAGE
02FD:
02FD:                 ; D_CONTROL debugging calls  These calls transfer data to the driver and
02FD:                 ; its I/O space from the user buffer  The format of the status list for these
02FD:                 ; calls is
02FD:                 ;
02FD:                 ;       80 : #bytes : disp : disp : data        Write to driver area
02FD:                 ;       81 : #bytes : disp :  00  : data        Write to C0Xx space
02FD:                 ;       82 : #bytes : disp :  00  : data        Write to CNxx space
02FD:                 ;       83 : #bytes : disp : disp : data        Write to C8xx space
02FD:                 ;
02FD:                 ;       #bytes - number of bytes to transfer, 00 to 255
02FD:                 ;
02FD:                 ; For various bizarre reasons, we choose to modify the store instruction
02FD:                 ; rather than use indexing  The range checking on the various calls depends
02FD:                 ; on how much code I write to do range checking
02FD:
02FD:                 ; Common code  Set up # bytes to transfer, bump CSLIST pointer, and
02FD:                 ; do the transfer  We do it in 1MhZ mode as we may be looking at the slot
02FD:
02FD: 20 5302         DCBx      JSR     DSCSET               ; go do setup
0300: 90**                      BCC     $2
0302:
0302:                 ; Setup barfed  Return error code in A
0302:
0302: 20 2819                   JSR     SysErr
0305:
0305: F0**            $2        BEQ     Leave                ; and scram if it's 00!
0307:
0307:                 ; Define the instruction as an abs STA (blecch!)
0307:
0307: A9 8D                     LDA     #8D
0309: 8D F902                   STA     Gak                  ; set up as an abs STA instruction!
030C:
030C:                 ; set 1MhZ mode, and do the transfer
030C:
030C:                           set1mhz
0317:
0317: B1 C3           DCloop    LDA     (CSLIST),Y           ; pick up user data
0319: 20 F902                   JSR     Gak                  ; put it away
031C: C8                        INY
031D: EE FA02                   INC     ADDRL
0320: D0**                      BNE     $1
0322: EE FB02                   INC     ADDRH
0325: C6 D0           $1        DEC     NBYTES               ; bump pointers, decrement count
0327: D0EE                      BNE     DCloop               ; loop through all bytes
0329:
0329:                           set2mhz                      ; back to full speed
0334: 60              Leave     RTS                          ; all done
0335:                           END

AB - Absolute     LB - Label       UD - Undefined    MC - Macro
RF - Ref          DF - Def         PR - Proc         FC - Func
PB - Public       PV - Private     CS - Consts

ADDRH    LB 02FB  ADDRL    LB 02FA  ALLOCSIR AB 1913  BADOP    LB 006F  BADREQ   LB 006A  BREAD    AB 00CB  BREG     AB FFEF
BUFFER   AB 00C2  BUMP     LB 01FD  CHAR     PB ----  CSLIST   AB 00C3  CSNO     LB 0194  CTLSTAT  AB 00C2  DC00     LB 01EE
DC01     LB 01EF  DC02     LB 01E0  DCBX     LB 02FD  DCB      LB 0020  DCFIN    LB 02E2  DCLOOP   LB 0317  DCLOSE   LB 00F5
DCONTROL LB 01C9  DEALCSIR AB 1916  DEBUG    LB 0022  DIS      LB 0000  DIBBLOCK LB 001A  DIBPTR   LB 0027  DIBSLOT  LB 0015
DINIT    LB 008D  DOIT     LB 0059  DOPEN    LB 00CA  DOTABLE  LB 0079  DREAD    LB 010E  DS00     LB 01BF  DS01     LB 018A
DS02     LB 01B8  DS80     LB 0273  DS81     LB 0288  DS82     LB 0298  DS83     LB 02CE  DSBX     LB 0218  DSCSET   LB 0253
DSLOOP   LB 0235  DSTATUS  LB 01A0  DWRITE   LB 0167  ENTRY    LB 0033  EREG     AB FFDF  FIXUP    LB 01F2  GAK      LB 02F9
GETBYTE  LB 0166  IMAT     MC ----  INCADR   MC ----  INITDM   LB 0024  INW      MC ----  LEAVE    LB 0334  NBYTES   AB 00D0
NGPARAM  LB 026F  NLCHAR   LB 0028  NLFLAG   LB 002A  NOTOPEN  LB 0074  OPENFLG  LB 0029  PUTBYTE  LB 019F  RDEND    LB 019A
REQCNT   AB 00C4  REQCODE  AB 00C0  RETCNT   AB 00D1  RLOOP    LB 0128  SCRAM    LB 0252  SELC800  AB 1922  SET1MHZ  MC ----
SET2MHZ  MC ----  SIRADDR  LB 002C  SIRCOUNT AB 0005  SIRTABLE LB 002B  SLOTCN   LB 0025  SLOTCX   LB 0026  SWFAR    LB [illegible]
SOSBLK   AB 00C6  SOSUNIT  AB 00C1  SWITCH   MC ----  SYSERR   AB 1928  WLOOP    LB 017E  XBADOP   AB 0026  XCTLCODE AB 0021
XCTLPARA AB 0022  XEOFERRD AB 004C  XIOERROR AB 0027  XNODRIVE AB 0028  XNORESRC AB 0025  XNOTAVAI AB 0024  XNOTOPEN AB 0023
XREQCODE AB 0020

Current minimum space is 20993 words.

Assembly complete:          905 lines
     0   Errors flagged on this Assembly
```

C

6502B Instruction Set

6502 Microprocessor Instructions

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right one Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index to Accumulator

The Following Notation Applies to this Summary:

A	Accumulator	$\veebar$	Logical Exclusive Or
X, Y	Index Registers	↑	Transfer From Stack
M	Memory	↓	Transfer To Stack
C	Borrow	→	Transfer To
P	Processor Status Register	←	Transfer To
S	Stack Pointer	V	Logical OR
√	Change	PC	Program Counter
—	No Change	PCH	Program Counter High
+	Add	PCL	Program Counter Low
$\wedge$	Logical AND	OPER	Operand
-	Subtract	#	Immediate Addressing Mode

FIGURE 1. ASL-SHIFT LEFT ONE BIT OPERATION

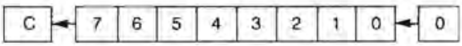


FIGURE 2. ROTATE ONE BIT LEFT (MEMORY OR ACCUMULATOR)

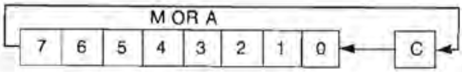
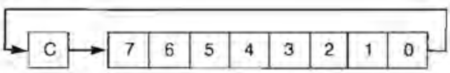


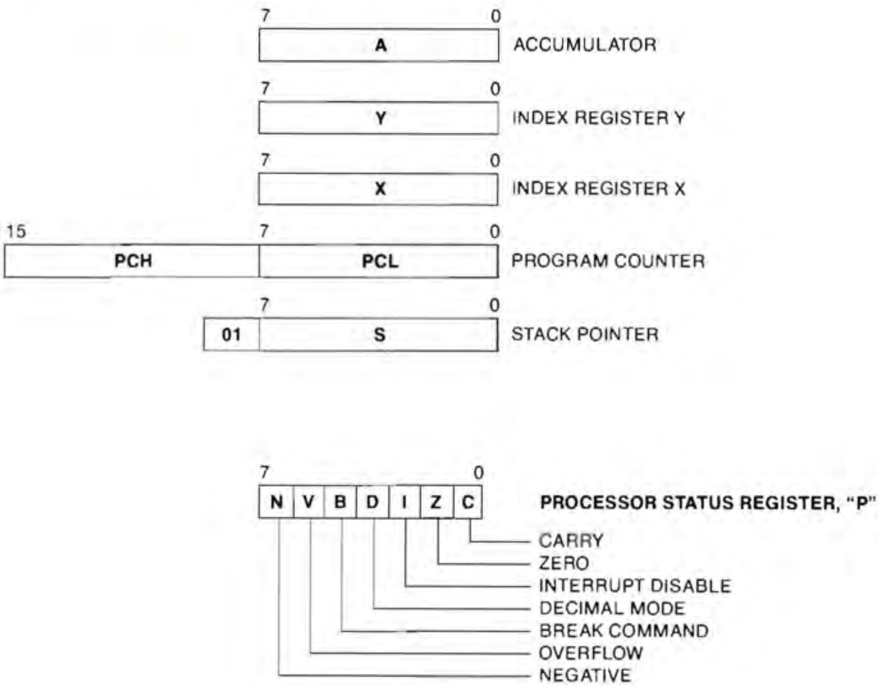
FIGURE 3.



NOTE 1: BIT — TESTS BITS

Bit 6 and 7 are transferred to the status register. If the result of A $\wedge$ M is zero then Z=1, otherwise Z=0.

Programming Model



Instruction Codes

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
ADC Add memory to accumulator with carry	A+M+C →A,C	Immediate	ADC #Oper	69	2	√√√--√
		Zero Page	ADC Oper	65	2	
		Zero Page,X	ADC Oper,X	75	2	
		Absolute	ADC Oper	6D	3	
		Absolute,X	ADC Oper,X	7D	3	
		Absolute,Y	ADC Oper,Y	79	3	
		(Indirect,X)	ADC (Oper,X)	61	2	
		(Indirect),Y	ADC (Oper),Y	71	2	
AND "AND" memory with accumulator	A Λ M →A	Immediate	AND #Oper	29	2	√√----
		Zero Page	AND Oper	25	2	
		Zero Page,X	AND Oper,X	35	2	
		Absolute	AND Oper	2D	3	
		Absolute,X	AND Oper,X	3D	3	
		Absolute,Y	AND Oper,Y	39	3	
		(Indirect,X)	AND (Oper,X)	21	2	
		(Indirect,Y)	AND (Oper),Y	31	2	
ASL Shift left one bit (Memory or Accumulator)	(See Figure 1)	Accumulator	ASL A	0A	1	√√√---
		Zero Page	ASL Oper	06	2	
		Zero Page,X	ASL Oper,X	16	2	
		Absolute	ASL Oper	0E	3	
		Absolute,X	ASL Oper,X	1E	3	
BCC Branch on carry clear	Branch on C=0	Relative	BCC Oper	90	2	------
BCS Branch on carry set	Branch on C=1	Relative	BCS Oper	B0	2	------
BEQ Branch on result zero	Branch on Z=1	Relative	BEQ Oper	F0	2	------
BIT Test bits in memory with accumulator	A Λ M, M_7 →N, M_6 →V	Zero Page	BIT* Oper	24	2	M_7√---M_6
		Absolute	BIT* Oper	2C	3	
BMI Branch on result minus	Branch on N=1	Relative	BMI Oper	30	2	------
BNE Branch on result not zero	Branch on Z=0	Relative	BNE Oper	D0	2	------
BPL Branch on result plus	Branch on N=0	Relative	BPL Oper	10	2	------
BRK Force Break	Forced Interrupt PC+2↓P↓	Implied	BRK*	00	1	---1--
BVC Branch on overflow clear	Branch on V=0	Relative	BVC Oper	50	2	------

Note 1 5 and 7 are transferred to the status register if the result of A V M is then 1 otherwise Z +0

Note 2 A BRK command cannot be masked by setting 1

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
BVS Branch on overflow set	Branch on V=1	Relative	BVS Oper	70	2	------
CLC Clear carry flag	0→C	Implied	CLC	18	1	---0--
CLD Clear decimal mode	0→D	Implied	CLD	D8	1	-0----
CLI	0→1	Implied	CLI	58	1	---0--
CLV Clear overflow flag	0→V	Implied	CLV	B8	1	0-----

CMP
Compare memory and accumulator
	A—M	Immediate
		Zero Page
		Zero Page,X
		Absolute
		Absolute,X
		Absolute,Y
		(Indirect,X)
		(Indirect),Y
			CMP #Oper
			CMP Oper
			CMP Oper,X
			CMP Oper
			CMP Oper,X
			CMP Oper,Y
			CMP (Oper,X)
			CMP (Oper),Y
				C9
				C5
				D5
				CD
				DD
				D9
				C1
				D1
					2
					2
					2
					3
					3
					3
					2
					2
						✓✓✓---

CPX Compare memory and index X	X—M	Immediate Zero Page Absolute	CPX #Oper CPX Oper CPX Oper	E0 E4 EC	2 2 3	✓✓✓---
CPY Compare memory and index Y	Y—M	Immediate Zero Page Absolute	CPY #Oper CPY Oper CPY Oper	C0 C4 CC	2 2 3	✓✓✓---
DEC Decrement memory by one	M—1→M	Zero Page Zero Page,X Absolute Absolute,X	DEC Oper DEC Oper,X DEC Oper DEC Oper,X	C6 D6 CE DE	2 2 3 3	✓✓----
DEX Decrement index X by one	X—1→X	Implied	DEX	CA	1	✓✓----
DEY Decrement index Y by one	Y—1→Y	Implied	DEY	88	1	✓✓----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
EOR "Exclusive-Or" memory with accumulator	A V M → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	EOR #Oper EOR Oper EOR Oper,X EOR Oper EOR Oper,X EOR Oper,Y EOR (Oper,X) EOR (Oper),Y	49 45 55 4D 5D 59 41 51	2 2 2 3 3 3 2 2	√√----
INC Increment memory by one	M + 1 → M	Zero Page Zero Page,X Absolute Absolute,X	INC Oper INC Oper,X INC Oper INC Oper,X	E6 F6 EE FE	2 2 3 3	√√----
INX Increment index X by one	X + 1 → X	Implied	INX	E8	1	√√----
INY Increment index Y by one	Y + 1 → Y	Implied	INY	C8	1	√√----
JMP Jump to new location	(PC+1) → PCL (PC+2) → PCH	Absolute Indirect	JMP Oper JMP (Oper)	4C 6C	3 3	------
JSR Jump to new location saving return address	PC+2↓ (PC+1) → PCL (PC+2) → PCH	Absolute	JSR Oper	20	3	------

LDA
Load accumulator with memory
	M → A	Immediate
		Zero Page
		Zero Page,X
		Absolute
		Absolute,X
		Absolute,Y
		(Indirect,X)
		(Indirect),Y
			LDA #Oper
			LDA Oper
			LDA Oper,X
			LDA Oper
			LDA Oper,X
			LDA Oper,Y
			LDA (Oper,X)
			LDA (Oper),Y
				A9
				A5
				B5
				AD
				BD
				B9
				A1
				B1
					2
					2
					2
					3
					3
					3
					2
					2
						√√----

LDX Load index X with memory	M → X	Immediate Zero Page Zero Page,Y Absolute Absolute,Y	LDX #Oper LDX Oper LDX Oper,Y LDX Oper LDX Oper,Y	A2 A6 B6 AE BE	2 2 2 3 3	√√----
LDY Load index Y with memory	M → Y	Immediate Zero Page Zero Page,X Absolute Absolute,X	LDY #Oper LDY Oper LDY Oper,X LDY Oper LDY Oper,X	A0 A4 B4 AC BC	2 2 2 3 3	√√----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
LSR Shift right one bit (memory or accumulator)	(See Figure 1)	Accumulator Zero Page Zero Page,X Absolute Absolute,X	LSR A LSR Oper LSR Oper, X LSR Oper LSR Oper,X	4A 46 56 4E 5E	1 2 2 3 3	0√√–––
NOP No operation	No Operation	Implied	NOP	EA	1	––––––
ORA "OR" memory with accumulator	A V M → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	ORA #Oper ORA Oper ORA Oper,X ORA Oper ORA Oper,X ORA Oper,Y ORA (Oper,X) ORA (Oper),Y	09 05 15 0D 1D 19 01 11	2 2 2 3 3 3 2 2	√√––––
PHA Push accumulator on stack	A ↓	Implied	PHA	48	1	––––––
PHP Push processor status on stack	P ↓	Implied	PHP	08	1	––––––
PLA Pull accumulator from stack	A ↑	Implied	PLA	68	1	√√––––
PLP Pull processor status from stack	P ↑	Implied	PLP	28	1	From Stack
ROL Rotate one bit left (memory or accumulator)	(See Figure 2)	Accumulator Zero Page Zero Page,X Absolute Absolute,X	ROL A ROL Oper ROL Oper,X ROL Oper ROL Oper,X	2A 26 36 2E 3E	1 2 2 3 3	√√√–––
ROR Rotate one bit right (memory or accumulator)	(See Figure 3)	Accumulator Zero Page Zero Page,X Absolute Absolute,X	ROR A ROR Oper ROR Oper,X ROR Oper ROR Oper,X	6A 66 76 6E 7E	1 2 2 3 3	√√√–––

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
RTI Return from interrupt	P↑PC↑	Implied	RTI	40	1	From Stack
RTS Return from subroutine	PC↑, PC+1→PC	Implied	RTS	60	1	------
SBC Subtract memory from accumulator with borrow	A − M − C → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	SBC #Oper SBC Oper SBC Oper,X SBC Oper SBC Oper,X SBC Oper,Y SBC (Oper,X) SBC (Oper),Y	E9 E5 F5 ED FD F9 E1 F1	2 2 2 3 3 3 2 2	√√√--√
SEC Set carry flag	1 → C	Implied	SEC	38	1	--1---
SED Set decimal mode	1 → D	Implied	SED	F8	1	----1-
SEI Set interrupt disable status	1 → I	Implied	SEI	78	1	---1--
STA Store accumulator in memory	A → M	Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	STA Oper STA Oper,X STA Oper STA Oper,X STA Oper,Y STA (Oper,X) STA (Oper),Y	85 95 8D 9D 99 81 91	2 2 3 3 3 2 2	------
STX Store index X in memory	X → M	Zero Page Zero Page,Y Absolute	STX Oper STX Oper,Y STX Oper	86 96 8E	2 2 3	------
STY Store index Y in memory	Y → M	Zero Page Zero Page,X Absolute	STY Oper STY Oper,X STY Oper	84 94 8C	2 2 3	------
TAX Transfer accumulator to index X	A → X	Implied	TAX	AA	1	√√----
TAY Transfer accumulator to index Y	A → Y	Implied	TAY	A8	1	√√----
TSX Transfer stack pointer to index X	S → X	Implied	TSX	BA	1	√√----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
TXA Transfer index X to accumulator	X → A	Implied	TXA	8A	1	√√----
TXS Transfer index X to stack pointer	X → S	Implied	TXS	9A	1	------
TYA Transfer index Y to accumulator	Y → A	Implied	TYA	98	1	√√----

Hex Operation Codes

00 — BRK
01 — ORA — (Indirect, X)
02 —
03 —
04 —
05 — ORA — Zero Page
06 — ASL — Zero Page
07 —
08 — PHP
09 — ORA — Immediate
0A — ASL — Accumulator
0B —
0C —
0D — ORA — Absolute
0E — ASL — Absolute
0F —
10 — BPL
11 — ORA — (Indirect), Y
12 —
13 —
14 —
15 — ORA — Zero Page, X
16 — ASL — Zero Page, X
17 —
18 — CLC
19 — ORA — Absolute, Y
1A —
1B —
1C —
1D — ORA — Absolute, X
1E — ASL — Absolute, X
1F —
20 — JSR
21 — AND — (Indirect, X)
22 —
23 —
24 — BIT — Zero Page
25 — AND — Zero Page
26 — ROL — Zero Page
27 —
28 — PLP
29 — AND — Immediate
2A — ROL — Accumulator
2B —
2C — BIT — Absolute
2D — AND — Absolute
2E — ROL — Absolute
2F —
30 — BMI
31 — AND — (Indirect), Y
32 —
33 —
34 —
35 — AND — Zero Page, X
36 — ROL — Zero Page, X
37 —
38 — SEC
39 — AND — Absolute, Y
3A —
3B —
3C —
3D — AND — Absolute, X
3E — ROL — Absolute, X
3F —
40 — RTI
41 — EOR — (Indirect, X)
42 —
43 —
44 —
45 — EOR — Zero Page
46 — LSR — Zero Page
47 —
48 — PHA
49 — EOR — Immediate
4A — LSR — Accumulator
4B —
4C — JMP — Absolute
4D — EOR — Absolute
4E — LSR — Absolute
4F —
50 — BVC
51 — EOR — (Indirect), Y
52 —
53 —
54 —
55 — EOR — Zero Page, X
56 — LSR — Zero Page, X
57 —
58 — CLI
59 — EOR — Absolute, Y
5A —
5B —
5C —
5D — EOR — Absolute, X
5E — LSR — Absolute, X
5F —
60 — RTS
61 — ADC — (Indirect, X)
62 —

63 —
64 —
65 — ADC — Zero Page
66 — ROR — Zero Page
67 —
68 — PLA
69 — ADC — Immediate
6A — ROR — Accumulator
6B —
6C — JMP — Indirect
6D — ADC — Absolute
6E — ROR — Absolute
6F —
70 — BVS
71 — ADC — (Indirect), Y
72 —
73 —
74 —
75 — ADC — Zero Page, X
76 — ROR — Zero Page, X
77 —
78 — SEI
79 — ADC — Absolute, Y
7A —
7B —
7C —
7D — ADC — Absolute, X NOP
7E — ROR — Absolute, X NOP
7F —
80 —
81 — STA — (Indirect, X)
82 —
83 —
84 — STY — Zero Page
85 — STA — Zero Page
86 — STX — Zero Page
87 —
88 — DEY
89 —
8A — TXA
8B —
8C — STY — Absolute
8D — STA — Absolute
8E — STX — Absolute
8F —
90 — BCC
91 — STA — (Indirect), Y
92 —
93 —
94 — STY — Zero Page, X
95 — STA — Zero Page, X
96 — STX — Zero Page, Y
97 —
98 — TYA
99 — STA — Absolute, Y
9A — TXS
9B —
9C —
9D — STA — Absolute, X
9E —
9F —
A0 — LDY — Immediate
A1 — LDA — (Indirect, X)
A2 — LDX — Immediate
A3 —
A4 — LDY — Zero Page
A5 — LDA — Zero Page
A6 — LDX — Zero Page
A7 —
A8 — TAY
A9 — LDA — Immediate
AA — TAX
AB —
AC — LDY — Absolute
AD — Absolute
AE — LDX — Absolute
AF —
B0 — BCS
B1 — LDA — (Indirect), Y
B2 —
B3 —
B4 — LDY — Zero Page, X
B5 — LDA — Zero Page, X
B6 — LDX — Zero Page, Y
B7 —
B8 — CLV
B9 — LDA — Absolute, Y
BA — TSX
BB —
BC — LDY — Absolute, X
BD — LDA — Absolute, X
BE — LDX — Absolute, Y
BF —
C0 — CPY — Immediate
C1 — CMP — (Indirect, X)
C2 —
C3 —
C4 — CPY — Zero Page
C5 — CMP — Zero Page
C6 — DEC — Zero Page
C7 —
C8 — INY
C9 — CMP — Immediate
CA — DEX
CB —
CC — CPY — Absolute
CD — CMP — Absolute
CE — DEC — Absolute
CF —
D0 — BNE
D1 — CMP — (Indirect), Y
D2 —
D3 —
D4 —
D5 — CMP — Zero Page, X
D6 — DEC — Zero Page, X
D7 —
D8 — CLD
D9 — CMP — Absolute, Y
DA —
DB —
DC —
DD — CMP — Absolute, X
DE — DEC — Absolute, X
DF —
E0 — CPX — Immediate
E1 — SBC — (Indirect, X)
E2 —
E3 —
E4 — CPX — Zero Page
E5 — SBC — Zero Page
E6 — INC — Zero Page
E7 —
E8 — INX
E9 — SBC — Immediate
EA —
EB —
EC — CPX — Absolute
ED — SBC — Absolute
EE — INC — Absolute
EF —
F0 — BEQ
F1 — SBC — (Indirect), Y
F2 —
F3 —
F4 —
F5 — SBC — Zero Page, X
F6 — INC — Zero Page, X
F7 —
F8 — SED
F9 — SBC — Absolute, Y
FA —
FB —
FC —
FD — SBC — Absolute, X
FE — INC — Absolute, X
FF —

D

Important Fixed Addresses

D
Important Fixed Addresses

There are several addresses that are commonly used by device drivers, entry points for SOS resources available to device drivers, and areas of memory that are often referred to.

SOS Resources Available for Device Driver's Use

ALLOCSIR	$1913	To allocate SOS Internal Resource
DEALCSIR	$1916	To deallocate SOS Internal Resource
SELC800	$1922	To select the $C800 address space for a given expansion slot
SYSERR	$1928	To report execution errors to SOS
QUEEVENT	$191F	To signal SOS that an event is to be queued

Addresses Important to Device Drivers

$FFD0	Zero-page (Z) Register
$FFDF	Environment (E) Register
$FFEF	Bank (B) Register
$18C0–C9	Driver parameter table area
$18CA–FF	Free zero-page area
$14C0–C9	Parameter table extend-page
$14CA–FF	Extend-page free area

Glossary

address *n.* A name or number designating a location in either the computer's memory or an on-line file.

algorithm *n.* Any mechanical or computational procedure.

analog data *n.* Data representable as fractional numbers.

analog-to-digital converter *n.* A device that converts measurements of continuously varying physical quantities such as temperature, voltage, or current into a digital form that can be used by a computer.

ASCII *n.* ASCII is an acronym for the American Standard Code for Information Interchange. This code assigns a unique value from 0 to 127 to each of 128 numbers, letters, special characters, and control characters.

assembler *n.* A program that converts assembly-language instructions into machine-language instructions.

assembly language *n.* A computer language made up of simple words, called mnemonics, that can be quickly and easily converted to machine language. Assembly-language programs are less difficult for people to write and understand than programs written in machine language.

binary *n.* The base-two numbering system consisting of the two digits, 0 and 1. Most computer storage devices are designed to store binary digits and computer circuitry is designed to manipulate information coded in a binary form.

bit *n.* Contraction of Binary digIT; the smallest amount of information that a computer can hold. A single bit specifies a single value of either "0" or "1". A group of 4 bits together form a nibble, 8 bits form a byte, and various numbers of bits form words.

board *n.* Short for printed-circuit board, or PC board. A sheet of material, usually made of fiberglass or phenolic-resin-impregnated paper. Attached to either or both faces and often even within the board are layers of copper, etched into the fine lines of specific circuits. Connected to these copper circuits are electronic components: resistors, capacitors, coils, and various solid-state devices.

bootstrap or **boot** *v.* To get the system running. The primitive bootstrap program loads into the computer a more sophisticated program that then takes over the responsibility for the overall operation of the computer.

buffer *n.* A device or area of memory that is allocated to hold information temporarily. Buffers act to generally speed up the performance of computer systems.

bus *n.* A group of wires that carry a related set of data, such as the bits of an address, in a standard format from one place to another. A bus can transmit information from one part of a computer to another, between the computer and a peripheral device, or between peripheral devices.

byte *n.* A basic unit of a computer's memory. A byte usually comprises eight bits and is thus capable of expressing a range of numbers from 0 to 255. (2 to the 8th power is 256.) Each character in the ASCII code can be represented in one byte, with an extra bit left over.

card ***n.*** A type of printed-circuit board that has a built-in connector so that it may be plugged into a larger board or other piece of hardware.

catalog ***n.*** See directory.

Central Processing Unit, or **CPU** ***n.*** The "brain" of the computer. The CPU is responsible for executing instructions that control the use of memory and perform arithmetic and logical operations. A microprocessor is a CPU.

character ***n.*** Any symbol that has a widely-understood meaning. In computers, letters, numbers, punctuation marks, and even what are normally just concepts, such as carriage returns, are all characters.

code ***n.*** 1. A computer program. 2. A method of representing something in terms of something else. The ASCII code represents characters as binary numbers; the BASIC and Pascal languages are codes that represent algorithms in terms of program statements.

cold start or cold boot ***v.*** To begin operation of the computer or a given program on the computer by loading in the operating system and the program, and then running.

command ***n.*** 1. An order given to the computer to perform an immediate action. 2. The part of an instruction that specifies the action to be carried out. In the BASIC instruction "PRINT "Hello" ", PRINT is the command. In the Pascal instruction "writeln ('Hello')", writeln() is the command.

compiler ***n.*** A program that translates a high-level language version of a program (the source code) into a low-level language version (the object code).

computer ***n.*** A machine that is controlled by stored instructions and is used to store, transfer, and transform information.

control character ***n.*** Control characters, the first thirty-two characters of ASCII, initiate, modify, or stop control functions.

controller ***n.*** See peripheral device controller.

CRT An acronym for Cathode-Ray Tube. A CRT is a tube with a phosphor-coated optical glass faceplate which, when struck by a directed beam of electrons generated within, glows with visible light. Some examples of CRTs are oscilloscope tubes, radar screens, and TV or monitor screens.

data ***n.*** Information that can be processed by a computer.

default ***n.*** The value or action selected by the system when the user does not select an allowable value or action.

delimiter ***n.*** A character that is used to designate the beginning or end of a string of characters and therefore is not considered a part of the string. Spaces are common delimiters of English words. /Computers/often/allow/other/symbols./

device ***n.*** A piece of computer hardware, such as a disk drive or terminal. Device is short for peripheral device.

device driver ***n.*** A small program that acts as a communications link between a device and the operating system.

digital data ***n.*** Data representable as whole numbers. See analog data.

directory ***n.*** A table of information about the files stored on a mass storage device such as a diskette. Information in a directory can include the length and address of files, the amount of storage space files occupy, etc.

disk ***n.*** A flat, circular piece of plastic (flexible disk) or metal (hard disk), either electroplated or coated with a fine magnetic powder, onto which information is magnetically recorded.

disk drive ***n.*** A device that can read information from and record information on a flexible disk or hard disk in much the same way that a tape recorder reads from and records on magnetic tape.

diskette ***n.*** The smaller (5 1/4 inch) of two usual forms of flexible disk (also called floppy disk), the other (8 inch) simply being called a flexible (or floppy) disk.

display 1. ***n.*** Information exhibited visually, especially on the screen of a display device. 2. ***v.*** To exhibit information visually. 3. ***n.*** A display device.

edit ***v.*** To change stored data or modify its format (for example, to insert, delete or move characters in a file).

editor ***n.*** A program that interacts with the user, allowing entry of text, graphics, and so on, into the computer and convenient editing of that information.

execute ***v.*** 1. To carry out a command or instruction. 2. (colloq.) To run a program or a portion of a program.

file ***n.*** A named, ordered collection of data.

file name ***n.*** The name used to identify a file. The operating system is able to locate that file by its name.

firmware ***n.*** Software stored in a ROM.

flexible disk ***n.*** See diskette.

floppy disk ***n.*** See diskette.

graphics ***n.*** 1. Information that is conveyed in terms of pictures (as distinguished from text). 2. Information displayed from a page of graphics memory, rather than text memory. Such a graphics page typically requires eight to sixteen times as much memory as a text page. This information might include text. An example would be an annotated chart or graph.

hardware ***n.*** The physical components of a computer and its peripheral devices.

Hertz (Hz) ***n.*** Cycles per second. A bicycle wheel which makes two revolutions in one second is spinning at a rate of 2 Hz. The Apple III's microprocessor runs at approximately 2 million Hz, or 2 MHz.

hexadecimal ***n.*** A number system which uses the ten digits 0 through 9 and the six letters A through F to represent values in base 16. Assembly-language instructions often use hexadecimal notation.

high-level language ***n.*** A programming language that is relatively easy for humans to understand. FORTRAN, BASIC, and Pascal are all examples of high-level languages. One statement of a high-level language usually corresponds to several statements in a low-level language.

I/O ***adj.*** Short for input/output: a general term referring to the transfer of information into and out of a computer or peripheral device.

I/O device ***n.*** An input/output device attached to a computer that makes it possible to bring information into the computer and for the computer to send information to the user or to another device. Such devices include keyboards, monitor screens, and serial interface cards.

IC ***n.*** See integrated circuit

input ***n.*** Information (data) arriving at a computer or device.
v. 1. The act of receiving data. 2. To type information into a computer. (jargon)

instruction ***n.*** The smallest portion of a program that a computer can execute. In 6502 machine language, an instruction comprises one, two, or three bytes and corresponds to a single machine operation. In a higher-level language, an instruction may be many characters long and may correspond to many operations.

integrated circuit (IC) ***n.*** A small piece (smaller than the size of a fingernail and about as thin) of pure, crystalline semiconductor material, usually silicon, that has had refined impurities introduced to form the various elements of an electronic circuit. Integrated circuits, or chips, are the basic building blocks of computers.

interface ***n.*** 1. The electronic components that allow two different devices, or the computer and a device to communicate. 2. The part of a computer program that interacts with the user.

interpreter ***n.*** A program, usually written in machine language, that individually translates each step in a high-level language program into a series of low-level machine language operations and then carries out those operations before proceeding to the next step. This is different from a compiler, which does all the translating before the resultant object code is run. The execution of an interpreted high-level program typically takes up to 100 times as long as that of compiled object code.

K ***n.*** A prefix (kilo), derived from Greek, used to denote one thousand. In common computer-related usage, K usually represents 2 to the 10th power or 1024.

kilobyte ***n.*** 1024 bytes.

load ***v.*** To transfer a program or data into the computer's memory.

low-level language ***n.*** Relative to high-level languages, low-level languages are simpler, more primitive, and are more tightly tied to the hardware of the computer than to the intuitive thought processes of a human. Low-level languages on Apple computers include assembly and machine languages.

machine language ***n.*** The computer language that controls the lowest-level internal operations of the computer. Each statement or instruction in machine language causes the machine to perform one operation.

memory ***n.*** Devices in which data can be stored and from which the data can be obtained at a later time. Typical memory devices include several types of integrated circuits (normally found within the computer), disks, punched cards (do not fold, spindle, or mutilate), and magnetic tapes. The information in a memory may be permanent, that is, it may be read from but not written to (see Read-Only Memory), or information may be written into as well as read from a memory (see read/write memory). Memory is further defined as to how specific locations of information may be accessed; there is Random-Access Memory and serial access memory.

microcomputer *n.* A computer that uses a microprocessor as the primary part of its Central Processing Unit.

microprocessor *n.* A Central Processing Unit contained in a single integrated circuit.

mnemonic *n.* A short, easy-to-remember word or group of letters that stands for another word. Assembly-language instructions are mnemonics.

monitor *n.* 1. A CRT, or CRT with its attendant circuits, which looks like a TV set with no channel selectors. 2. A computer program that allows the user to directly initiate machine-language instructions.

native code *n.* The machine-language code usable directly by the CPU of the computer upon which the code is to be run. See P-code and P-machine.

network *n.* 1. A number of interconnected, individually controlled computers. 2. The hardware system used to interconnect such a group of computers.

object code *n.* The code that results from a program's source code, written in a high-level language, being translated by a compiler or assembler into a lower-level language.

operating system *n.* The collection of programs that organize a computer and its peripheral devices into a working unit that can be used to develop and execute applications programs.

output *n.* Data that have been, are being, or are to be transmitted. *v.* The act of transmitting data. (jargon)

page *n.* 1. A screenful of information on a video display. A page is not necessarily 8.5" x 11". 2. A segment of internal storage.

peripheral *n.* A shortened form of "peripheral device". A device attached to the computer that can provide input and/or accept output from the computer. Peripherals include printers, disk drives, and speech synthesizers.

peripheral device controller ***n.*** A specialized circuit that connects a peripheral device to the computer. Such controllers are called intelligent if they include small device handlers held in ROMs. Controllers for the Apple II computer are most easily used if intelligent; those for the Apple III use software device handlers that are stored on diskette and become part of the operating system.

P-code ***n.*** Short for pseudo-code. Program instructions intended to be executed by a P-machine.

P-machine ***n.*** Short for pseudo-machine. Software that emulates a CPU. P-machines are created to allow one computer to imitate the CPU of another and thus to run software created for that other computer's CPU. (Purists will point out that some P-machines imitate CPUs that don't really exist at all.) Programs run on a P-machine run slower than they would if the hardware CPU of the computer could run them directly.

port ***n.*** The point of connection between the computer and peripheral devices, other computers, or a network. A port is usually a physical connector terminating a bus.

program ***n.*** A stored sequence of instructions that causes a computer to perform some function or operation. ***v.*** To create such a sequence of instructions.

protocol ***n.*** A set of conventions governing information exchange between two communicating computers, or between a computer and a peripheral device.

Random-Access Memory (RAM) ***n.*** 1. Memory that has a unique address for each unit of storage and a method by which each unit may be immediately read from or written to. Such memory is made up of some minimum grouping of bits; either nibbles, bytes, or words. 2. The integrated circuits forming the main read-write memory of the computer. The values stored in most types of RAM memories are lost when power is no longer supplied.

Read-Only Memory (ROM) *n.* The integrated circuits that contain the computer's permanent memory; phonograph records and optical disks are ROMs. Information stored in ROM is not lost when the power is removed. Most ROM is randomly accessible, but the term random-access memory is usually reserved for read-write memory that is randomly accessible.

read-write memory *n.* Memory in which values may be stored and read by the processor. Random-Access Memory, magnetic tape, and disks are each read-write memories.

scroll *v.* To move all the information on a display (usually upward) to make room for more information (usually at the bottom of the screen).

software *n.* A collective term for computer programs. Software is generally stored for future use on either disk or magnetic tape. When actually being executed, software is typically held in read-write memory.

SOS (Sophisticated Operating System) *n.* The operating system used by the Apple III computer. It is designed to allow easy development of new languages and the addition of new peripheral devices while maintaining compatibility with existing hardware and software running under SOS.

source code *n.* The original version of a program, written in a high-level language for later compilation or assembly.

word *n.* A group of bits that occupies one storage location and is treated by the operating system as a unit and is transported as such. A word is differentiated from both a byte (8 bits) and a nibble (4 bits) in that its length is defined by the underlying design of the CPU being used. Apple computer CPUs typically use either 1- or 2-byte words. See P-machine.

Figures and Tables

1 Overview of SOS Device Drivers

2 The Physical Environment of SOS

3 Request Handling

4 SOS-provided Services

5 Interrupt Handling

7 Interfacing with Apple III Peripheral Connectors

Index