Apple

Device and Interrupt Support Tools Manual

Apple II Pascal 1.2

WORK-
BENCH

030-0248-A

Table of Contents

Preface — Read Me First — xi

xi What You Will Find Here
xiii Hardware and Software You Will Need
xiii Where to Look for More Information

Chapter 1 — Introduction to Device Drivers — 3

3 Runtime Support for the Pascal System
4 The Purpose of a Device Driver
5 Types of Devices Supported by Pascal
6 Device Numbers
7 Attaching Drivers at Run Time
8 The Three Attach Files
9 What SYSTEM.ATTACH Does

Chapter 2 — Writing a Device Driver — 11

11 General Guidelines
13 Returning an IORESULT
14 Maintaining Type-Ahead
15 Device Driver Subroutines
15 Initialize
16 Read and Write
19 Status
22 Transient Initialization

Chapter 3 Attaching a Device Driver 25

25 The Four Steps
26 Using ATTACHUD
31 Utility Programs for Handling Attach Data
31 Using the File Merge Utility (ADMERG)
32 Using the Listing Utility (SHOWAD)
33 Using the Conversion Utility (CONVAD)
34 Error Messages You Might Encounter at Startup Time

Chapter 4 Interrupt Management 37

37 Handling Interrupts in Apple II Pascal
39 Sequence for All Systems but the IIc
40 Sequence for the Apple IIc
42 Writing an Interrupt-Based Driver
42 Enabling and Disabling Interrupts
43 Initializing the Chain of Interrupt Service Routines
44 Recognizing an Interrupt
44 Returning From an Interrupt Service Routine
45 Automatic UNITCLEAR
45 Environmental Considerations
47 Attaching an Interrupt-Based Driver
48 Installing the IM
48 On a 128K Pascal System
49 On a 64K Pascal System

Chapter 5 Special BIOS Considerations 51

51 Calling BIOS Routines
53 The Interpreter's Jump Table
54 BIOS Jump Tables
57 Returning to the Pascal Interpreter
58 Handling Special Characters
60 Managing the Type-Ahead Buffer
61 Permanent Locations Used by the BIOS and Interpreter

Appendix A — *Sample Code for a Device Driver* — 65

Appendix B — *Sample Code for an Interrupt-Based Driver* — 69

Appendix C — *Information Contained in ATTACH.DATA* — 79

Appendix D — *Peripheral Card Firmware Protocols* — 83

Appendix E — *Drivers for Standard Pascal System Devices* — 87

- 87 Character Devices
  - 89 CONSOLE: and SYSTERM:
  - 92 PRINTER:
  - 92 REMIN: and REMOUT:
  - 93 Special Use of Unit #3
- 93 Block-Structured Devices
- 94 Sample Code for a CONSOLE: Driver

Appendix F — *IORESULT Codes* — 99

Index — 101

List of Figures and Tables

Chapter 1 — Introduction to Device Drivers — 3

4	Figure 1-1	Runtime Support
6	Table 1-1	Device Names and Numbers

Chapter 2 — Writing a Device Driver — 11

18	Figure 2-1	The MODE Parameter for UNITREAD and UNITWRITE
20	Figure 2-2	The CONTROL Parameter for UNITSTATUS

Chapter 4 — Interrupt Management — 37

38	Figure 4-1	A Chain of Interrupt Service Routines

Chapter 5 — Special BIOS Considerations — 51

52	Figure 5-1	Bank Switching for the BIOS
55	Table 5-1	Order of Addresses in BIOSAF
61	Table 5-2	Zero Page Locations
62	Table 5-3	$BF00 Page Locations
63	Table 5-4	$FF00 Page Locations

Read Me First

This manual is a guide to writing and attaching your own device drivers for use with the Apple® II Pascal 1.2 operating system.

Before reading it, you should be familiar with the changes to Apple II Pascal described in the *Apple II Pascal 1.2 Update*.

Since a device driver is an assembly-language program, you should also be familiar with the Apple II Pascal 6502 Assembler. This is described in Chapter 6 of the *Apple Pascal Operating System Reference Manual* for the Apple II.

What You Will Find Here

Here is an overview of what this manual contains.

- Chapter 1 provides general information about device drivers—what they are meant to do, where they reside at run time, and what types of devices they support. It also gives a general overview of how devices are attached.
- Chapter 2 gives guidelines for writing device driver code.
- Chapter 3 describes in detail how to attach a device driver to the Pascal operating system.
- Chapter 4 explains interrupt handling in Apple II Pascal and the additional requirements for device drivers that are meant to handle interrupts.

- Chapter 5 contains technical information about the Pascal system's BIOS, including the mechanisms for calling both standard and user-defined device drivers. You may want to skim this chapter on a first reading, but you should be familiar with the information it contains before you attempt to assemble and attach a user-defined driver.
- Sample code for device drivers is provided in Appendix A and Appendix B. It may be useful to refer to this code while you are writing your own driver.
- Appendix C lists the information contained in an attach data file, and gives the format in which it is saved.
- Appendix D summarizes the protocol for peripheral card firmware to communicate with programs in Apple II Pascal, version 1.2.
- Appendix E describes the special requirements of standard Pascal system device drivers. You will want to read through it if the device driver you plan to write is intended to replace one of the system device drivers.
- Appendix F lists IORESULT codes to be returned by device driver subroutines.

Look for these aids to understanding throughout the manual:

Gray Boxes: Gray boxes contain useful pieces of related information.

Warning
Warning boxes point out potential problems or disasters.

Hardware and Software You Will Need

You should use this manual with the following software and hardware:

- The *Attach Tools* disk, provided with this manual. This disk contains the following programs, which you will need to use in writing and attaching drivers:

  SYSTEM.ATTACH
  ATTACHUD.CODE
  ADMERG.CODE
  CONVAD.CODE
  SHOWAD.CODE
  IM.CODE

Warning
The version of SYSTEM.ATTACH shipped on the Attach Tools disk is for use with 64K and 128K Apple II Pascal 1.2 Development Systems only. The version shipped with Apple II Pascal 1.2 Runtime Systems is for use with Runtime Systems only. These two versions of SYSTEM.ATTACH are not *interchangeable.*

- Apple II Pascal, version 1.2. This manual does not apply to earlier versions of the Pascal operating system.
- An Apple II, Apple II Plus, Apple IIe, or Apple IIc.

If you are already using a user-defined driver that was created under Apple II Pascal 1.1, you must convert your ATTACH.DATA file to the version 1.2 format. To do this, use the CONVAD utility described in Chapter 3. In addition, a few of the memory locations used by the BIOS have been changed in version 1.2. If the driver relies on any of these, you will have to modify it accordingly. Refer to the tables at the end of Chapter 5 to see if this affects you.

Where to Look for More Information

For technical information about the Apple II, Apple II Plus, or Apple IIe, see the *Apple IIe Reference Manual*. For technical information about the Apple IIc, refer to the *Apple IIc Reference Manual*.

For technical information about bank-switching, see the *Apple Language Card Installation and Operation Manual* for the Apple II.

Chapter 1

Introduction to Device Drivers

A *device driver* is an assembly-language program that performs basic input/output operations on a peripheral device. A driver can also contain code to handle interrupts.

The Apple II Pascal operating system comes with drivers built in for a console (video monitor and keyboard), one or more flexible disk drives, a printer, and a remote device such as a phone line with a modem. If you want Apple II Pascal to drive another device, you must write your own device driver to control it.

This chapter explains device driver basics: how they are accessed at run time, what they are meant to do, what kinds of devices they drive, how those devices are referenced, and how they are attached.

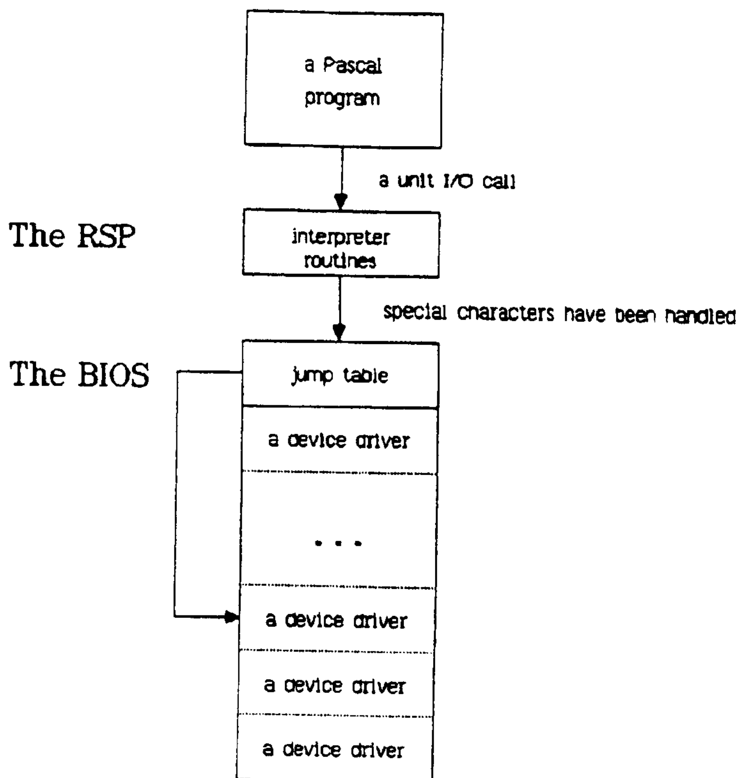
Runtime Support for the Pascal System

A portion of the Apple II Pascal system is present in main memory whenever programs are run. This code includes a group of interpreter routines called the Runtime Support Package (RSP).

The RSP does some initial processing of input/output calls, and then calls the Basic Input/Output Subsystem (BIOS). The BIOS is a major portion of the code that is resident at run time, and includes all of the Pascal system's device drivers.

This scheme is represented in Figure 1-1.

Figure 1-1. *Runtime Support*



The device drivers that are built into the Pascal system are always present in the BIOS. User-defined device drivers are supplied in a file on your startup disk, and are attached to the BIOS at startup time. Chapter 3 describes this process in detail.

The Purpose of a Device Driver

A device driver can perform four basic input/output (I/O) operations:

- initializing the device
- reading data from the device
- writing data to the device
- returning status information about the device

A driver need not perform all four operations. For instance, the printer driver writes data but does not read it.

These operations correspond to the Pascal *unit I/O* procedures UNITCLEAR, UNITREAD, UNITWRITE, and UNITSTATUS.

- UNITCLEAR resets the specified device and returns a Pascal IORESULT that indicates whether the call was successful.
- UNITREAD and UNITWRITE transfer data from (or to) the specified device.
- UNITSTATUS returns status information about the specified device. It can also be used to send control commands to the device.

When a Pascal program makes a standard I/O call such as READ or WRITELN, this is translated into a unit I/O call.

The unit I/O procedures can be used to access any I/O device that is attached to the Pascal system.

Types of Devices Supported by Pascal

The Pascal system supports two types of input/output devices: *character* devices and *block-structured* devices.

A character device is a device that sends or receives a stream of ASCII characters. (Not all of the characters need to be printable.) Some character devices, such as the console, both send and receive characters. Other character devices, such as the printer, can only receive characters.

A block-structured device is a disk drive. For the purposes of the Pascal system, disks are organized into consecutive *blocks* of data. A block contains 512 bytes.

At the hardware level, disks are actually organized into sectors and tracks. On the Disk II, for example, a block consists of two sectors, and there are 280 consecutive blocks on the disk, numbered from 0 to 279.

The device driver for a block-structured device must be able to transfer one or more blocks. The driver's code is responsible for translating Pascal system block numbers into the disk hardware's track-and-sector addresses.

Device Numbers

Every device—both standard and user-defined—attached to the Pascal system is referenced by a *device number* (also called a *unit number*). When a program calls a unit I/O procedure, it must pass the number of the device it is to access.

Table 1-1 shows the numbers used to specify devices in the Pascal system.

Table 1-1. *Device Names and Numbers*

Device Number	Device Name	Description of the Device
#0:		(Not used)
#1:	CONSOLE:	Screen and keyboard with echo
#2:	SYSTERM:	Keyboard without echo
#3:		(Not used)
#4:	<disk name>:	Startup disk drive
#5:	<disk name>:	Second disk drive
#6:	PRINTER:	Printer
#7:	REMIN:	Remote input
#8:	REMOUT:	Remote output
#9:	<disk name>:	Fifth disk drive
#10:	<disk name>:	Sixth disk drive
#11:	<disk name>:	Third disk drive
#12:	<disk name>:	Fourth disk drive
#13:		A block-structured user-defined device
...		...
#20:		A block-structured user-defined device
#128:		ProFile
#129:		Mouse
#130:		A user-defined device
...		...
#143:		A user-defined device

User-defined devices can use device numbers 13-20 (for block-structured devices only) and 128-143 (for devices of any kind).

Warning
If you assign device numbers 128 and 129 to user-defined devices, you will not be able to use the drivers for the ProFile or the Mouse.

Devices with numbers 128-143 can be accessed *only* by calls to UNITCLEAR, UNITREAD, UNITWRITE, and UNITSTATUS.

Attaching Drivers at Run Time

Before a program can access a user-defined device, the device's driver must be *attached* to the Pascal operating system. This ensures that the device driver will be present in the BIOS in main memory and thus made available to programs at run time.

When a user-defined driver is attached, its code is placed on the Pascal system heap in a region that is not accessible to ordinary Pascal programs. The advantages of this scheme include the following:

- Software vendors can write device drivers that will be loaded at startup time and will be invisible to the user.
- Drivers cannot be lost through improper heap management, since the drivers are loaded on the heap in the operating system's space before any user program is allowed to allocate heap space.
- The system is not restricted to using any special (hard-coded) memory locations, since device driver code is relocatable.

The only disadvantage is that a user-defined device driver does occupy memory space that might otherwise be used by a Pascal code file.

The Three Attach Files

To attach your own device driver, you must make sure three special files—ATTACH.DRIVERS, ATTACH.DATA, and SYSTEM.ATTACH—are present on your startup disk.

- ATTACH.DRIVERS contains the actual code for your user-defined device drivers. It is a library file and is created by using Pascal's LIBRARY utility program.
- ATTACH.DATA contains a data record for each driver in ATTACH.DRIVERS. This information is used by SYSTEM.ATTACH. It is created by running the program ATTACHUD (short for ATTACH User Device), described in detail in Chapter 3.
- SYSTEM.ATTACH is the program that does the work of attaching user-defined device drivers to the BIOS. When SYSTEM.ATTACH is present on your startup disk, it is automatically executed by the Pascal operating system at startup time.

What SYSTEM.ATTACH Does

For each user-defined device, SYSTEM.ATTACH does the following things (roughly in this order):

- Loads the code for the driver into memory.
- Provides the BIOS with the address of the new driver.
- Does a UNITCLEAR on the new device (this is optional for some devices).
- Loads and executes the driver's transient initialization code.

The process of attaching a device driver is more involved if the driver uses interrupts. For details, refer to Chapter 4, which deals with interrupt management.

In general, SYSTEM.ATTACH initializes each device by calling the device driver's UNITCLEAR routine. This initialization is optional for user-defined devices with numbers 128-143, but devices that use interrupts must always be initialized by SYSTEM.ATTACH. Devices attached to device numbers 1-20 are always initialized by SYSTEM.ATTACH.

When the Pascal system is restarted (due to a system error or a user interrupt), *all* devices are reinitialized by a UNITCLEAR call. This includes user devices that are not initialized by SYSTEM.ATTACH at startup time.

Chapter 2

Writing a Device Driver

A device driver must be written in accordance with certain guidelines if the Pascal system is to properly handle a user-defined device. This chapter goes into detail about what is required of a user-defined device driver. It starts with general guidelines and then gets specific about code to return IORESULTS, maintain type-ahead, and perform the four major subroutines of a device driver.

If your device driver is meant to support interrupts, then some additional requirements apply. These are discussed in Chapter 4.

As you read this chapter, you may wish to refer to Appendix A, which contains an example of a user-defined device driver.

Chapter 5 contains additional information on the BIOS and its format. You may need to refer to this material while you are writing your device driver.

If you intend to write a device driver to replace one of the standard Pascal system device drivers, see Appendix E for additional requirements for these standard drivers.

General Guidelines

A device driver must be written in assembly language using the Pascal 6502 Assembler, described in Chapter 6 of the *Apple Pascal Operating System Reference Manual* for the Apple II.

The code for the device driver must be relocatable, so you *must not* assemble your driver using the .ABSOLUTE directive. Relocatable code allows the system to manage memory properly after the driver has been loaded.

The device driver typically contains at least four subroutines—one each to handle INIT, READ, WRITE, and STATUS types of calls. Sometimes not all these subroutines are implemented. For example, a READ call is unnecessary for a write-only device such as a printer. If one of these calls is unimplemented, the device driver can simply pop the parameters from the stack, load 0 into the X Register (this indicates no error), and then return.

There must be *one* entry point for the device driver, which means it must contain one and only one .PROC directive.

The driver's common entry point is the one used by the Pascal system. When a program calls UNITCLEAR, UNITREAD, UNITWRITE, or UNITSTATUS, the system jumps to the code at the .PROC location. The X Register (XREG) is set to a code that indicates what type of call this is—an INIT, READ, WRITE, or STATUS call—and the A Register is set to the device number (the UNITNUMBER parameter).

For this reason, your device driver must examine the X Register to determine which type of call has been made. The possible values of XREG are as follows:

XREG	=	0	means	READ	(no bits are set)
		1		WRITE	(bit 0 is set)
		2		INIT	(bit 1 is set)
		4		STATUS	(bit 2 is set)

Once the value of XREG has been tested, your driver must jump to the appropriate subroutine.

When the driver is called, the top of the stack contains a return address. The driver must pop this address off the stack and save it; when it returns, it must first push this word on the stack, and then do an RTS.

Other parameters may be on the stack as well. These are discussed below, in the section on Device Driver Subroutines.

Returning an IORESULT

When the driver returns, it must also pass a completion code—an IORESULT—in the XREG. Appendix F contains a complete list of IORESULT codes that your driver can use.

The following IORESULT values are the ones your device driver is most likely to use.

IORESULT = 0 No error

If your device driver does not detect any errors, it must clear the XREG before it returns.

IORESULT = 3 Illegal I/O request

This should be used, for example, if your driver receives a UNITREAD call for a write-only printer.

IORESULT = 9 Volume not found

Return this error if the driver does not find the device's card in the slot where the driver expected it (see Appendix D).

IORESULT = 16 Disk is write protected

Use this for a UNITWRITE to a block-structured device when the disk is write-protected.

IORESULT = 17 Illegal block number

If a UNITREAD or UNITWRITE to a block-structured device specifies a block number that cannot be mapped to a disk address, use this error number. This number should also be returned when a UNITWRITE to a block-structured device would go beyond the end of the disk.

IORESULT = 128 through 255 (Available)

These values can be used for errors that are particular to the device your driver controls. The Pascal program that uses the device will have to detect the error and take appropriate action.

Other IORESULT values are used by standard Pascal system device drivers or other portions of the Pascal system, and your device driver doesn't need to deal with them.

Maintaining Type-Ahead

If you want the Pascal system to maintain the type-ahead buffer while your device driver is in use, your driver must call the BIOS routine CONCK before it does a read or a write. (CONCK stands for CONsole ChecK, and is pronounced "concheck.") This routine checks CONSOLE: to see if a character has been typed. If one has, it is placed in the type-ahead buffer, and CONCK returns.

Here is the code for a subroutine that calls CONCK and then returns to your driver (this also appears in Appendix A).

```
CONCKAD    .EQU 2              ;Declare label for the CONCK
                               ;address

CKR        .WORD  CONCKRT-1
GOTOCK     LDY    #55.         ;Load YREG with offset to
                               ;CONCK call
           LDA    @OE2,Y       ;Get the actual (current)
                               ;CONCK address
           STA    CONCKAD      ;Save it
           INY
           LDA    @OE2,Y       ;Get next byte of the CONCK
                               ;address
           STA    CONCKAD+1    ;Save it
           LDA    CKR+1        ;Set up for the return to
           PHA                 ;CONCKRT after the CONCK call
           LDA    CKR
           PHA
           JMP    @CONCKAD     ;Jump to CONCK in the BIOS

CONCKRT    RTS                 ;Return to caller (the READ
                               ;or WRITE routine)
```

Warning
The usual way to call CONCK is through the CONCKVECTOR vector located at $BF0A (see Chapter 5). Do not *do this in your driver—the BIOS will lose its return address to Pascal, and will not be able to return from your driver.*

Device Driver Subroutines

This section describes the purpose of each of the four main subroutines in a device driver.

Pascal provides the interfaces to the unit I/O, but it does *not* provide the implementation. The implementation (in terms of some specific device) is the task of a specific device driver contained in the BIOS. This means that whenever you call UNITCLEAR, UNITREAD, UNITWRITE, or UNITSTATUS, in effect you are calling routines in the BIOS directly.

For the standard devices supported by the Pascal system, the device driver code is already present in the BIOS (as we have mentioned). For a user-defined device, you must write the driver code yourself, and this code must conform to the interfaces specified by the Pascal language (and the Pascal system). Once you have written this code, you must attach it to the BIOS, as described in Chapter 3.

Initialize

The initialize subroutine must be called when the device driver receives a call from UNITCLEAR (XREG = 2).

A Pascal call to UNITCLEAR has this form:

UNITCLEAR (UNITNUMBER);

When UNITCLEAR is called, the top of the stack contains the return address. This is a one-word (two-byte) parameter. There are no other parameters on the stack.

This subroutine must do whatever is necessary to initialize the device's hardware, and then set the XREG to the appropriate IORESULT (0 means no error).

Initializing at Startup Time: When you start up your system, SYSTEM.ATTACH does the work of attaching your device driver to the BIOS. You can have SYSTEM.ATTACH call the initialize subroutine (and thus initialize the device) whenever you start up your Pascal system. To do this, answer `Y` to the prompt `Do you want this unit to be initialized at boot time?` when you run ATTACHUD (described in Chapter 3). Devices that use interrupts *must* be initialized by SYSTEM.ATTACH at startup time. Devices with numbers 1-20 are always initialized by SYSTEM.ATTACH.

Read and Write

The read subroutine must be called when the device driver receives a call from UNITREAD (XREG = 0), and the write subroutine must be called when the device driver receives a call from UNITWRITE (XREG = 1).

The Pascal calls to these subroutines have the following form:

UNITREAD (UNITNUMBER, ARRAY,
  LENGTH [, [BLOCKNUMBER] [,MODE]]);

UNITWRITE (UNITNUMBER, ARRAY,
  LENGTH [, [BLOCKNUMBER] [,MODE]]);

(Brackets indicate optional parameters.)

For both of these subroutines, the parameters are the same. These are all one-word (two-byte) parameters. When UNITREAD or UNITWRITE is called, the stack contains these parameters in the following order:

top of stack —>	return address	
	BLOCKNUMBER	
	LENGTH	(byte count)
	ARRAY	(buffer address)
	UNITNUMBER	(this is also in AREG when the device driver is called)
	MODE	(control word)

These are all one-word (two-byte) parameters.

The BLOCKNUMBER parameter contains the number of the first block to transfer (for a block-structured device). The value of BLOCKNUMBER will be 0 if the device is a character device, or if the parameter wasn't present in the call from Pascal.

The LENGTH parameter contains the number of bytes to be transferred.

The ARRAY parameter contains the starting address of the buffer.

The read subroutine must transfer bytes from the device to the buffer, starting at the buffer address that is on the stack. The number of bytes to transfer is in the LENGTH parameter. If the device is block-structured, then the BLOCKNUMBER parameter must be translated into a hardware disk address; bytes are read beginning at that address.

Warning

In the Pascal source program, if the value of LENGTH is greater than the declared size of the ARRAY parameter, then a UNITREAD can destroy data in memory. ARRAY should be a packed array that contains at least LENGTH bytes.

The write subroutine must transfer bytes from the buffer to the device, starting at the buffer address that is on the stack. The number of bytes to transfer is in the LENGTH parameter. If the device is block-structured, then the BLOCKNUMBER parameter must be translated into a hardware disk address; bytes are transferred to the location that begins at that address.

Transferring a Partial Block: If the number of bytes to transfer in a UNITWRITE call is not a multiple of 512 (the number of bytes in a block), the last block written will be only partially filled. It is not defined whether the remainder of the block is left intact or filled with garbage. If it is more convenient to write a full block, you can safely do so.

If the number of bytes to transfer in a UNITREAD call is not a multiple of 512, the last block may contain garbage that must *not* be transferred into main memory. If it is transferred, there is a danger of destroying data. If your driver must always read a full block from disk, then buffer the block within your driver, and then transfer exactly the number of bytes requested into the read buffer.

The UNITNUMBER parameter is the number of the device, and is provided for reference. If the driver is to handle only one device, then this parameter may not be needed. If the driver is to handle more than one of the same kind of device, then the driver can use this parameter to determine which device to access.

The MODE parameter is used (or can be used) as a control word. If the device is a character device, then the RSP uses this value to control two special write options. One is to insert a line-feed character (ASCII LF) after every end-of-line character (ASCII CR) that it encounters. Some devices, such as certain printers, require this.

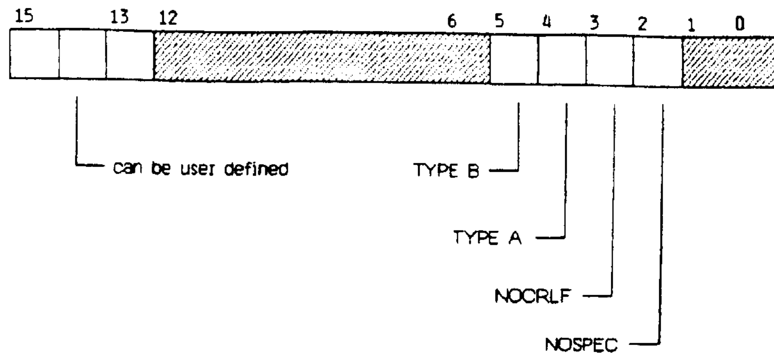
The other option is to expand the blank-compression code that can appear at the beginning of a line in a text file created by the Pascal system Editor. This code consists of two bytes: the first

byte is an ASCII DLE character (decimal 16), and the second byte contains the value 32 (decimal) plus the number of blanks that appear at the beginning of the line.

There is no defined use for this parameter in a call to UNITREAD, or when the device is block-structured. If the MODE parameter wasn't present in the call from Pascal, its value will be 0.

Figure 2-1 shows the defined format of the MODE parameter.

Figure 2-1. *Format of the MODE Parameter for UNITREAD and UNITWRITE*



Shaded areas are reserved.

Bits 13 . . 15 can be defined by the driver and its application.

If bit 2, NOSPEC, equals 1, do not expand DLE codes.

If bit 3, NOCRLF, equals 1, do not append LF to each CR.

If bit 4 equals 1, do not process TYPE A characters.

If bit 5 equals 1, do not process TYPE B characters.

As shown in the figure, bit 2, NOSPEC, controls the expansion of DLE codes, and bit 3, NOCRLF, controls the end-of-line/line feed option. These bits are processed by the RSP.

Bits 4 and 5 have to do with the processing of certain special characters. These bits are also processed by the RSP. See Chapter 5 for further details.

Bits 13-15 are available to your device driver. In a call to UNITREAD or UNITWRITE, they can be used to specify any special options that you wish to support for the device you are handling. Of course, any Pascal program that uses your device driver will have to supply the correct value(s) of MODE.

The remaining bits in the MODE parameter are reserved, and must not be used.

Status

The status subroutine must be called when the device driver receives a call from UNITSTATUS (XREG = 4).

A Pascal call to UNITSTATUS has the following form:

UNITSTATUS (UNITNUMBER, PAB, CONTROL);

When UNITSTATUS is called, the stack contains the parameters in the following order:

top of stack ->	return address	
	PAB	(buffer address)
	CONTROL	

These are all one-word (two-byte) parameters.

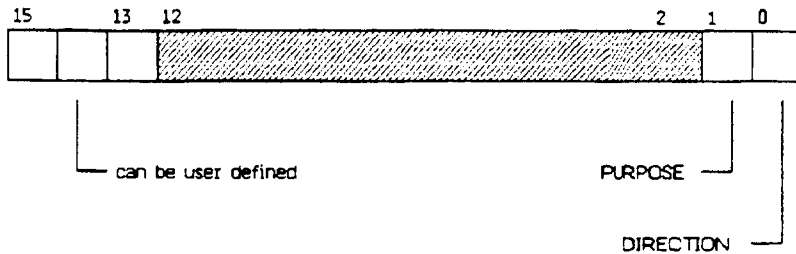
The PAB parameter contains the address of a packed array of bytes. This is a variable parameter that UNITSTATUS uses to return information. The size of PAB depends on the use of UNITSTATUS, and can vary from driver to driver. The Pascal program and the device driver *must* assume the same size for PAB; otherwise data can be lost when UNITSTATUS writes to memory.

Note that the UNITNUMBER parameter is not pushed onto the stack in a UNITSTATUS call, although its value is present in the A Register.

The status subroutine must examine the CONTROL parameter to see what sort of status information is requested, and then write out the appropriate values to the buffer that begins at the address specified on the stack.

Figure 2-2 shows the defined format of the CONTROL parameter.

Figure 2-2. *Format of the CONTROL Parameter for UNITSTATUS*



Shaded areas are reserved.

Bits 13 . . 15 can be defined by the driver and its application.

Bit 0, DIRECTION --

0 - return status of output channel

1 - return status of input channel

Bit 1, PURPOSE --

0 = device status requested

1 = device control requested

As shown in the figure, bit 0, DIRECTION, indicates whether the request was for the status of output (DIRECTION = 0) or input (DIRECTION = 1).

If the device you are handling is input-only or output-only, you can choose to have UNITSTATUS do nothing (that is, pop the parameters and clear the XREG) for a call that requests information about the wrong direction, or you can have it return an IORESULT of 3: Illegal I/O request.

Bit 1, PURPOSE, normally indicates that the subroutine should return status information (PURPOSE = 0). You can write your driver so that the STATUS subroutine also controls the device (PURPOSE = 1) in some way(s).

Bits 13-15 are available to your device driver. They can be used to specify any special options that you wish to support for the device you are handling. Of course, any Pascal program that uses your device driver will have to supply the correct value(s) of CONTROL.

The remaining bits in the CONTROL parameter are reserved, and must not be used.

In the Pascal system standard device drivers, UNITSTATUS is implemented only for standard disk devices (device numbers 4, 5, and 9-12) and for the CONSOLE:. See Appendix E for further details.

If you choose not to implement UNITSTATUS for your device, this subroutine should simply pop the parameters, clear the XREG (IORESULT of 0), and then return.

Transient Initialization

As we mentioned in Chapter 1, SYSTEM.ATTACH initializes devices by doing a UNITCLEAR on each device at startup time. (Devices with numbers 128-143 are not necessarily initialized—you can specify whether they are or not when you run ATTACHUD, described in detail in Chapter 3.)

SYSTEM.ATTACH has another initialization step, which is optional, but may be useful for some device drivers. A device driver can be accompanied by a *transient initialization module* that is executed *only* at startup time.

Transient initialization was provided for use by the Pascal ProFile driver, but it is available to any user-defined device driver.

After SYSTEM.ATTACH loads all device drivers onto the heap, and calls UNITCLEAR for each driver (if that is specified in the driver's attach data record), it loads and executes the transient initialization modules. Transient initialization modules are loaded and executed in the same order as their associated drivers.

A transient initialization module is executed immediately after it is loaded. The next transient initialization module overlays the previous one, so this code *goes away* as soon as it has been executed.

Transient initialization modules are prepared in the same way as device drivers. A transient initialization module must be written in assembly language, must be relocatable (that is, it must not include the .ABSOLUTE directive), and must have a single entry point (that is, one and only one .PROC directive).

The transient initialization module must be assembled separately from its associated device driver. Once it has been assembled, it must be placed in the ATTACH.DRIVERS file by using the LIBRARY utility.

When you run ATTACHUD, it asks you whether the driver has an associated transient initialization module. If you answer Y, ATTACHUD asks some further questions about the module. Like a device driver, a transient initialization module can be aligned on some particular byte boundary (from 0-256) in order to help structure data.

Note that the transient initialization code is executed *after* the device driver's own (callable) initialization code is executed.

When SYSTEM.ATTACH calls a transient initialization module, the top of the stack contains the address of the module's associated device driver. This allows the transient initialization module to call subroutines and use data contained within the device driver itself.

Note: The address is the actual .PROC location of the driver, and not (.PROC - 1).

Chapter 3

Attaching a Device Driver

This chapter explains how to attach your device driver with the three special files described briefly in Chapter 1—ATTACH.DRIVERS, ATTACH.DATA, and SYSTEM.ATTACH. It begins by outlining the four-step process for attaching drivers. It then goes into more detail about executing ATTACHUD, the utilities available for handling attach data files, and error messages you might encounter at startup time.

The Four Steps

Here is how you attach a new device driver, in four steps:

1. Write the device driver and assemble it, according to the guidelines in Chapter 2 (and Chapter 4 as well, if the driver uses interrupts).
2. Execute ATTACHUD, shipped on the *Attach Tools* disk, to create an attach data file for your driver. Name the file ATTACH.DATA and transfer this file to your startup disk.
3. Next, execute LIBRARY.CODE to place your driver's code in the ATTACH.DRIVERS file. Transfer this file to your startup disk. The LIBRARY utility is described in Chapter 8 of the *Apple Pascal Operating System Reference Manual* for the Apple II.
4. Transfer SYSTEM.ATTACH to your startup disk from the *Attach Tools* disk.

Warning
The version of SYSTEM.ATTACH that is shipped on the Attach Tools *disk is for use with 64K and 128K Apple II Pascal 1.2 Development Systems only. The version shipped with Apple II Pascal 1.2 Runtime Systems is for use with the Runtime Systems only. These two versions of SYSTEM.ATTACH* are not *interchangeable.*

When you start the Pascal system with SYSTEM.ATTACH, ATTACH.DRIVERS, and ATTACH.DATA present, SYSTEM.ATTACH loads your new driver, and you can then test it and use it.

Viewing File Contents: You can use the SHOWAD utility described later in this chapter to view the contents of an ATTACH.DATA file, and the LIBRARY utility to view the contents of ATTACH.DRIVERS.

Using ATTACHUD

ATTACHUD is an interactive program. When you execute it, it presents you with a series of questions about the device driver you have written, and uses your responses to create an attach data record. The output of ATTACHUD is a file that contains one or more attach data records.

If You Change Your Driver Code: If you change your device driver and reassemble it, you don't always need to run ATTACHUD a second time. Changes to driver code don't affect the data record in ATTACH.DATA *unless* you have changed something that affects the answer to one of ATTACHUD's questions. (Of course, you still need to use LIBRARY to place the new code in the ATTACH.DRIVERS file.)

When you execute ATTACHUD, it opens the session by displaying this prompt:

```
ATTACH.DATA Creation Utility [1.2]
Copyright Apple Computer, Inc. 1983

What will be the name of the attach data file?
(<RETURN> to exit program):
```

Type in the name you have chosen for the output file. If this is the first user-defined device driver in your system, call the new attach data file ATTACH.DATA.

If you have already defined one or more device drivers and created attach data files for them, call the new attach data file something else, like NEWDEVICE.DATA. Then execute the ADMERG utility described later in this chapter to append your driver data file NEWDEVICE.DATA to the existing ATTACH.DATA file.

ATTACHUD Error Messages: If at any point ATTACHUD can't handle your response, it displays a message in the form

```
ERROR => some error message
Please try again ...
(<RETURN> to exit program):
```

Decide what mistake you made, and then retype your response to the previous prompt. If you simply press RETURN, you terminate ATTACHUD. Several prompts allow you to cut a session short by pressing RETURN; each of them tells you so.

After you enter the name of the output file for this session, ATTACHUD asks you

```
Will you use the 2000.3FFF HIRES page? (Y/N)
```

A note informs you that these next questions will determine if any attached drivers can reside in the HiRes pages. Type Y for yes or N for no.

The next question is

```
Will you use the 4000.5FFF HIRES page? (Y/N)
```

Again, answer Y or N. Answer yes to these questions if you will ever be running a program that uses the named HiRes graphics page while your driver is attached.

Warning
Only a very large device driver will overlap a HiRes page. However, if you should answer no to one or both of these prompts, and a program does *use the HiRes page in which a driver has been loaded, that device will suddenly become unavailable and other messy bugs may crop up. Be cautious!*

After the questions about the HiRes pages, ATTACHUD asks:

```
What is this device driver's name?
It will be the assembly's .PROC name.
(<RETURN> to exit program):
```

Type in the name of the driver's .PROC entry point (see Chapter 2). This name cannot be more than eight characters long.

The next prompt is:

```
Assign what unit number to this driver?
Valid units are 1, 2, 4..20, 128..143
(<RETURN> to exit program):
```

Type in the device number (unit number) that you wish your programs to use when handling the device by calls to UNITCLEAR, UNITREAD, UNITWRITE, and UNITSTATUS. Refer to Table 1-1 for valid device numbers.

Block-structured devices can use numbers in the range 4, 5, 9-20; character devices can use numbers in the range 1, 2, 6-8; and devices of either kind can use numbers in the range 128-143. Typically, user-defined block-structured devices should use only device numbers 13-20. User-defined character devices should use only device numbers 130-143.

Warning

If you use the number of an existing system device (for example, 1 for CONSOLE:), then ATTACHUD accepts it, and SYSTEM.ATTACH goes ahead and loads the new driver into memory. This is dangerous unless you know what you are doing. System devices have their own requirements and peculiarities, and if you accidentally replace a system driver with a driver that does not work, some very serious bugs can occur. Probably the Pascal system will have to be restarted. The requirements of the standard Pascal device drivers are described in Appendix E.

After asking for the device number, ATTACHUD asks:

```
Do you want this unit to be initialized
at boot time? (Y/N)
```

If you answer Y, SYSTEM.ATTACH does a UNITCLEAR on this device whenever you start up your Pascal system. A driver that uses interrupts *must* be initialized at startup time.

ATTACHUD then asks:

```
Do you want another unit number to
refer to this device driver? (Y/N)
```

As many device (unit) numbers as desired can be attached to the driver. This can be a way to save memory space when you have more than one of the same kind of device attached to your Apple II. The devices should be either all character devices or all block devices.

If you answer `N` to this question, ATTACHUD skips ahead to the next prompt. If your driver has been written to drive more than one device and you answer `Y`, ATTACHUD takes you back to the `Assign what unit number ...` prompt.

The next prompts have to do with whether SYSTEM.ATTACH should load your driver so that it is aligned on a particular byte boundary.

```
Do you want this driver aligned on
a particular byte boundary? (Y/N)
```

If you answer `Y`, ATTACHUD displays the following:

```
The boundary can be between 0 and 256.
0=>Driver can start anywhere. (default)
8=>Driver starts on 8 byte boundary.
N=>Driver starts on N byte boundary.
256=>Driver starts on PAGE boundary.
What boundary do you want? (0..256)
(<RETURN> to exit program):
```

Type in the number that you wish SYSTEM.ATTACH to use. Byte alignment can make it easier for your driver's assembly code to access its internal data structures.

The next prompt asks about transient initialization code:

```
Do you want this driver to have a
transient initialization section? (Y/N)
```

If you answer `N`, ATTACHUD skips ahead to the question about interrupts. If you answer `Y`, the next question is

```
What is the transient's name?
It will be the assembly's .PROC name.
(<RETURN> to exit program):
```

This is just like the question about the name of the driver itself. See Chapter 2 for a description of transient initialization.

Next, ATTACHUD asks if the transient initialization code should be aligned on a byte boundary. These prompts are just like the prompts for the driver itself:

```
Do you want this transient aligned on
a particular byte boundary? (Y/N)
```

If you answer Y, the following prompt appears:

```
The boundary can be between 0 and 256.
0=>Transient can start anywhere. (default)
8=>Transient starts on 8 byte boundary.
N=>Transient starts on N byte boundary.
256=>Transient starts on PAGE boundary.
What boundary do you want? (0..256)
(<RETURN> to exit program):
```

The next prompt is

```
Will this driver use interrupts? (Y/N)
```

If your driver does use interrupts, answer Y. ATTACHUD then ensures that a data record for the interrupt manager (IM) is present at the end of the output file. See Chapter 4 for a discussion of interrupt handling. Be sure that if you answer Y to this, you previously answered Y to `Do you want this unit to be initialized at boot time?`

Finally, ATTACHUD asks you

```
Do you want to attach another driver? (Y/N)
```

If you want to attach another user-defined driver, answer Y. ATTACHUD takes you back to the `What is the name of this driver?` prompt.

Otherwise, answer N. ATTACHUD displays the following message to show that your session is through:

```
Attach data creation complete
```

Utility Programs for Handling Attach Data

After you have created an attach data file with ATTACHUD, you may want to examine or modify the file. Three utility programs on the *Attach Tools* disk let you do so.

- ADMERG lets you merge two or more attach data files.
- SHOWAD lists the contents of an attach data file.
- CONVAD converts records in an attach data file from Pascal 1.1 format to Pascal 1.2 format.

Using the File Merge Utility (ADMERG)

The ADMERG (for Attach Data MERGe) utility enables you to combine two (or more) attach data files into one. ADMERG creates a new file with one of each of the data records from the original files, and ensures that no record is duplicated.

Warning
ADMERG only allows one data record per device number. That is, if ADMERG has read a data record with a certain device number, then that record will be written to the output file, but if ADMERG encounters a second record with the same number, that second record will not *be written out. It is a good idea to use SHOWAD to examine the contents of each attach data file you plan to merge and make sure device numbers are not duplicated before you run ADMERG.*

When you execute ADMERG, it displays the following prompt:

```
Apple Pascal ATTACH.DATA File Merge Utility [1.2]
Copyright Apple Computer, Inc. 1981,1983

Enter name of NEW file to create (<RETURN> to exit
program):
```

Type the name of the new file you wish to create. If this is the file that will be used by SYSTEM.ATTACH, you must call it ATTACH.DATA.

ADMERG then prompts

```
Enter name of file to get Attach.data records FROM
(<RETURN> to exit program):
```

Type the name of one of the files that you want to merge.

Once ADMERG has processed that file, it asks

```
Another Input file? (Y/N)
```

If you answer Y, ADMERG asks for the name of the next file to merge, and so forth. Once you answer N to this prompt, ADMERG asks

```
Create another NEW file? (Y/N)
```

If you wish to create another new attach data file, answer Y. ADMERG returns you to the first prompt. Otherwise, ADMERG signals that you are done by displaying

```
Program terminated
```

Using the Listing Utility (SHOWAD)

The SHOWAD (for SHOW Attach Data) utility displays the contents of the records in an attach data file in an easily readable format.

When you execute SHOWAD, it displays this prompt:

```
Apple Pascal ATTACH.DATA Listing Utility [1.2]
Copyright Apple Computer, Inc. 1983

Enter name of ATTACH.DATA file (<RETURN> to exit
program):
```

Type the name of the attach data file you wish to view (it may have some name other than ATTACH.DATA).

If there is some problem in reading the file, SHOWAD displays

```
file is empty
```

Otherwise, SHOWAD displays the contents of each data record, in a format much like the following:

```
Driver Name - GISMET - Not Aligned
Attached to #130
Unit #'s to init at boot time - 130
This driver CANNOT be placed in the first HiRes graphics
space.
This driver CANNOT be placed in the second HiRes
graphics space.
This driver DOES use interrupts.
Driver does not have transient initialization code.
```

This tells you that the name of the driver (its .PROC entry point) is GISMET, that the driver is not aligned to any byte boundary, and that the GISMET driver is referenced by device number 130 (which would be used in a call to UNITCLEAR, UNITREAD, UNITWRITE, or UNITSTATUS).

The display also tells you that this driver cannot occupy either of the HiRes graphics pages, that it uses interrupts, and that it does not have any associated transient initialization code.

If the driver were aligned, the display would show this in the format

```
- Aligned on a 16 byte boundary
```

showing the appropriate number of bytes.

If the driver did have transient initialization code, this would be displayed as follows:

```
Name of Driver's transient initialization code - TRAN
Transient initialization code not aligned.
```

SHOWAD shows a similar display for every record present in the file that it reads.

Using the Conversion Utility (CONVAD)

The CONVAD (for CONVert Attach Data) utility simply converts the records in an attach data file from Pascal version 1.1 format to Pascal version 1.2 format.

If you never attached a device driver under Pascal version 1.1, you won't need to use this utility.

If you do have one or more attach data files that were created under Pascal version 1.1, then you *must* convert them by running CONVAD. Otherwise, the associated drivers will not be attached when you start up your version 1.2 Pascal system.

When you execute CONVAD, it displays this prompt:

```
Apple Pascal ATTACH.DATA Conversion Utility [1.2]
Copyright Apple Computer, Inc. 1983

Enter name of OLD (1.1) ATTACH.DATA file (<RETURN> to
exit program):
```

Type the name of the original attach data file. CONVAD then displays

```
Enter name of NEW (1.2) ATTACH.DATA file (<RETURN> to
exit program):
```

Type the name of the new file. If you foresee wanting to use the original file for some reason, don't use the same name as the old file when you answer this prompt.

Error Messages You Might Encounter at Startup Time

Several errors can occur when SYSTEM.ATTACH is executed at startup time. Following are the error messages that might be displayed, and what can be done to correct the error.

```
ERROR => No records in ATTACH.DATA
```

You must run ATTACHUD to create an attach data record for each driver in ATTACH.DRIVERS. This error can also occur if ATTACH.DATA was created under Apple Pascal version 1.1. In that case, convert it to version 1.2 format using the CONVAD utility.

```
ERROR => Reading segment dictionary of ATTACH.DRIVERS
```

There is something wrong with the library file ATTACH.DRIVERS. It may be possible to correct this by using the LIBRARY utility.

```
ERROR => Reading driver
```

There is something wrong with the library file ATTACH.DRIVERS, or possibly something is wrong with the driver code itself.

```
ERROR => A needed driver is not in ATTACH.DRIVERS
```

There is a data record in ATTACH.DATA that does not correspond to any driver in ATTACH.DRIVERS. The code for the driver must be placed in ATTACH.DRIVERS by using the LIBRARY utility.

```
ERROR => ATTACH.DATA needed by SYSTEM.ATTACH
```

The ATTACH.DATA file is not present on the startup disk.

```
ERROR => ATTACH.DRIVERS needed by SYSTEM.ATTACH
```

The ATTACH.DRIVERS file is not present on the startup disk.

```
ERROR => The Interrupt Manager (IM) driver is not in
ATTACH.DRIVERS
```

One or more of the ATTACH.DATA records specifies that its corresponding driver uses interrupts, but IM.CODE has not been placed in ATTACH.DRIVERS. (This error can occur only in 64K Pascal systems.) IM.CODE must be placed in ATTACH.DRIVERS using the LIBRARY utility.

```
ERROR => Reading transient
```

There is something wrong with the library file ATTACH.DRIVERS, or possibly something is wrong with the transient initialization code itself.

```
ERROR => A needed transient is not in ATTACH.DRIVERS
```

A data record in ATTACH.DATA specifies a transient initialization module that is not present in ATTACH.DRIVERS. The transient code must be placed in ATTACH.DRIVERS by using the LIBRARY utility.

Warning
If SYSTEM.ATTACH itself is not present on your startup disk, there will be no error message. The Pascal system will not even attempt to attach any user-defined device drivers, and when it has started up it will support only the standard devices.

Chapter 4

Interrupt Management

One or more devices that generate interrupts can be supported by Apple II Pascal, version 1.2. The portion of the Pascal system that controls interrupts is called the Interrupt Manager (IM). When you write a device driver for an interrupt-based device, you must meet the requirements of the IM, in addition to following the guidelines in Chapter 2.

The first section in this chapter describes interrupt handling in Apple Pascal. The sections that follow discuss how to write a device driver that supports interrupts, and how it must be attached. As you read this chapter, you may wish to refer to the sample code for an interrupt-based driver, which appears in Appendix B.

Note: 48K Runtime Systems do not support interrupts.

Handling Interrupts in Apple II Pascal

The main task in handling interrupts is to save the context of the current program and then restore that context once the interrupt has been processed. This includes saving the contents of various system registers and restoring them once the driver returns.

When an interrupt can come from one of several devices, it is also necessary to identify which device generated it, so that the appropriate driver can handle the interrupt.

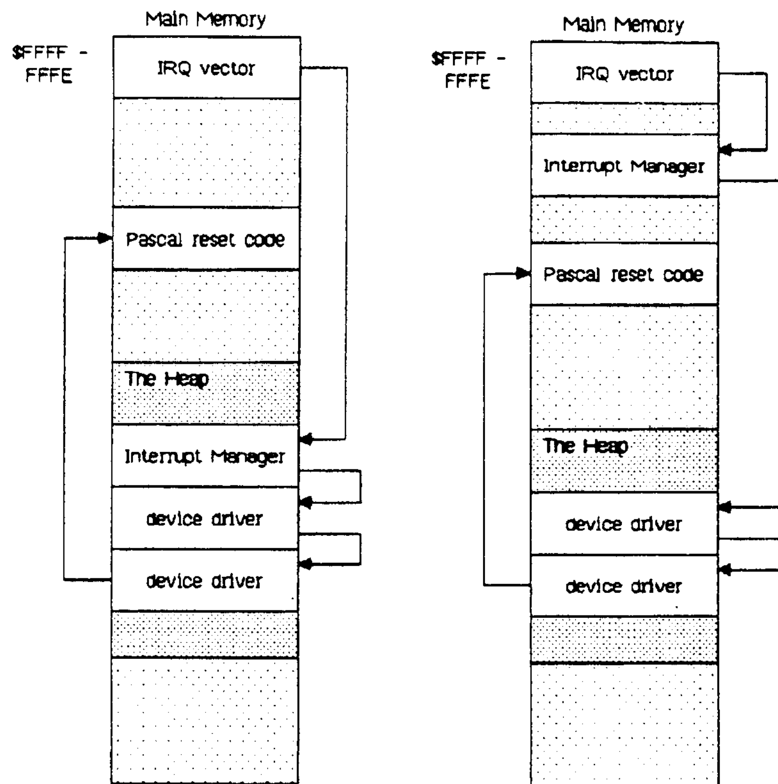
A driver for a device that supports interrupts must contain a section of code called the *interrupt service routine*. This code is called by the Interrupt Manager.

The Interrupt Manager (IM) itself is responsible for saving the current context and restoring it later. The interrupt service routines are responsible for determining whether they should handle a given interrupt. Just how they do this depends on the particular device.

Interrupt service routines are set up in a linked chain, as shown in Figure 4-1. Although this figure does not apply precisely to the Apple IIc, the overall scheme for the IIc is not essentially different.

Figure 4-1. *A Chain of Interrupt Service Routines*

Image on left applies to 64K Pascal system; image on right to 128K Pascal system



If an interrupt service routine recognizes an interrupt, it processes the interrupt and then returns to the IM. If the service routine doesn't recognize an interrupt, it transfers control to the next interrupt service routine in the chain. If none of the service routines claims an interrupt, then an error has occurred, and the system is restarted.

To determine whether it should process an interrupt, an interrupt service routine can (in general) check the interrupt flag register for the appropriate card slot.

The location of the interrupt flag register may vary according to the hardware; it is best if the peripheral card follows the conventions described in the *Apple IIe Design Guidelines* manual, in the section on Peripheral Card Firmware. See Appendix D for a summary of these guidelines.

For 64K Pascal systems, the code for the IM is in the form of an ATTACH driver. However, the IM cannot be called from a user program. For 128K Pascal systems, interrupt handling is built in, and the IM code is ignored if it is present in ATTACH.DRIVERS. See the section below on Installing the IM.

Sequence for All Systems but the IIc

For all systems but the Apple IIc, interrupts are handled in the following sequence:

1. A device interrupt occurs. This disables interrupts. If the interrupt was generated by a BRK instruction, the processor sets the break bit in the processor status byte.
2. The processor then jumps to the code whose address is stored in $FFFE-FFFF. This is the Interrupt Manager.
3. If the break bit is set, the IM restarts the Pascal system. Otherwise, the IM saves the current context and then transfers control to the first interrupt service routine in the chain.

4. If the service routine doesn't recognize the interrupt, it transfers control to the next service routine in the chain. Otherwise, it processes the interrupt and then returns to the IM.
5. If the last interrupt service routine in the chain doesn't recognize the interrupt, it transfers control to the reset code for the Pascal system.
6. When the IM regains control, it restores the interrupted program's context, which re-enables interrupts. Execution proceeds from the point at which it was interrupted.

A spurious interrupt can be generated as the result of a hardware malfunction, or of a BRK instruction in currently executing code. In the case of a hardware malfunction, the interrupt falls through the chain of routines, and control is ultimately passed to the Pascal system reset code. In the case of a BRK instruction, the processor sets the break bit and the IM then restarts the Pascal system.

Sequence for the Apple IIc

The sequence of events described above does not apply to the Apple IIc because the IIc's firmware performs some of the tasks that are handled by the IM on other versions of the Apple II.

Interrupt Handler Address: The address of the Apple IIc's built-in interrupt handler is always in $FFFE-FFFF of the Monitor ROM, and is automatically stored in $FFFE-FFFF of both language cards when the Pascal system starts up.

Here is the sequence of events on an Apple IIc:

1. A device interrupt occurs. This disables interrupts. If the interrupt was generated by a BRK instruction, the processor also sets the break bit in the processor status byte.
2. The processor then jumps to the code whose address is stored in the IRQ vector ($FFFE-FFFF).
3. The IRQ vector points to the built-in interrupt handler in the IIc firmware (this is in the $C800 space).

4. The IIc firmware saves the current context and sets the context to a known state, as follows:

   If 80STORE and PAGE2 are on, then text page 1 is switched in so that the main screen holes are accessible (the PAGE2 soft switch is turned off).

   Main memory is switched in for reading (RAMRD soft switch).

   Main memory is switched in for writing (RAMWRT soft switch).

   Monitor ROM ($D000 through $FFFF) is switched in for reading.

   The main stack and zero page are switched in (ALTZP soft switch).

   The auxiliary stack pointer is preserved, and the main stack pointer is restored.

5. The firmware checks to see if the break bit was set. If so, the firmware jumps to the code whose address is stored in $03F0-03F1. Since this is the location of the Pascal system's type-ahead buffer, the results are random, and the Pascal system must be restarted.

6. If the break bit was not set, the firmware checks to see if the interrupt is one that it knows how to handle (for example, some Mouse interrupts are handled completely by the firmware). If this is the case, the firmware handles the interrupt.

7. If the interrupt is not a standard firmware interrupt, the firmware passes the interrupt to the Pascal system by jumping to the code whose address is stored in $03FE-03FF (in main memory).

8. The address contained in $03FE-03FF is $03F4, so the code at $03F4 is executed. This code switches in the main language card, write-enables it, and then jumps to a special location in the IM.

9. The IM then transfers control to the first interrupt service routine in the chain.

10. If the service routine doesn't recognize the interrupt, it transfers control to the next service routine in the chain. Otherwise, it processes the interrupt and returns to the IIc firmware via an RTI instruction.
11. When the firmware interrupt handler regains control, it restores the original context of the system, and then does an RTI to resume processing from where it was when the interrupt occurred.
12. If the last interrupt service routine in the chain doesn't recognize the interrupt, it transfers control back to the start of the IIc interrupt handler in the firmware. This, of course, causes the system to loop indefinitely!

More detailed information about interrupt handling on the Apple IIc can be found in the *Apple IIc Reference Manual*.

Writing an Interrupt-Based Driver

This section discusses the considerations you must take into account when you write a driver for a device that generates interrupts.

An interrupt service routine must be an integral part of your device driver's code. This ensures that it will be loaded by SYSTEM.ATTACH. If you don't do this, your code is in danger of being released by the system, and a subsequent interrupt may cause unpredictable effects.

Sample code for an interrupt-based device driver appears in Appendix B. You may wish to refer to this appendix as you read through this section.

Any device number (1-20 or 128-143) can be used. Do not replace a standard device number unless you are familiar with the information in Appendix E. The IM itself is assigned the highest available number.

Enabling and Disabling Interrupts

SYSTEM.ATTACH enables interrupts after the full chain of interrupt service routines has been built, and all transient initialization modules have been executed. Device drivers (or devices, in their firmware) must *never* re-enable interrupts if they have been disabled by the IM.

In addition, if you wish to execute some code with interrupts disabled, this should not be done with just an SEI instruction. Instead you should use the sequence of PHP, SEI <code> PLP. This ensures that the system state is correctly restored when you exit the critical section (after the PLP).

Interrupts are disabled when an interrupt occurs, and the IM re-enables interrupts after the interrupt has been serviced. Only one interrupt can be handled at a time. Interrupts are also disabled by the Pascal system during disk accesses.

On the Apple IIe, interrupts are periodically disabled while 80-column screen operations are being performed. This is most noticeable while the display is scrolling. Also, most Apple IIe peripheral cards disable interrupts during read and write operations, and re-enable them when reading or writing is done.

Initializing the Chain of Interrupt Service Routines

Any driver that uses interrupts must initialize itself before the system starts up in order to link its interrupt service code into the chain of service routines. The initialization code should do the following (before exiting) in order to initialize the links:

```
          LDA   OFFFE       ;Move IRQ vector into next
          STA   STOREIT     ;driver pointer
          LDA   OFFFF
          STA   STOREIT+1

          LDA   I_ADDRESS   ;Move int service routine
          STA   OFFFE       ;address into the IRQ vector
          LDA   I_ADDRESS+1
          STA   OFFFF

I_ADDRESS .WORD I_HANDLER
STOREIT   .WORD 0           ;Next driver pointer
```

where I_HANDLER is the entry point of the driver's interrupt service routine, and STOREIT will contain the address of the next interrupt service routine to be called if the current one finds that its device did not generate the interrupt.

This code must be executed only *once* and must *not* be in a transient initialization module. The driver itself can also contain regular initialization code to reset the device or its buffer, and so forth (see the section on Automatic UNITCLEAR, below).

Recognizing an Interrupt

At the start of its interrupt service routine(s), a device driver must first determine whether the driver's device hardware generated the interrupt. In general, this involves checking a register on the device's controller card (for example, an interrupt flag register on a 6522), but the details are device-dependent

If the interrupt was generated by the driver's device, the driver should process the interrupt and then return to the IM by an RTI instruction.

If the interrupt was not generated by the driver's device, the driver should do an indirect jump to the next device driver. (The address of the next driver is saved as STOREIT in the sample initialization code shown in the previous section.) If this device driver is the last in the chain, the jump will be to the Pascal system reset code.

This jump is accomplished automatically, since the system initializes the IRQ vector to point to the reset code. If the initialization for all interrupt-based device drivers follows the scheme shown above and in Appendix B, then this pointer will be moved to the end of the interrupt service routine chain. Note that this does not apply to the IIc.

If Device Card Can't Signal Interrupts: If your device card has no way of signaling that it generated an interrupt, then its service routine *must* be the last service routine in the chain. It will have to assume that if it is called, it will handle an interrupt. Since the routine won't be able to detect a hardware failure interrupt, this is not a good approach, and should be avoided if at all possible.

To ensure that a driver is the last one in the interrupt drivers chain, assign it a unit number lower than all other interrupt driver unit numbers.

Returning From an Interrupt Service Routine

At the end of an interrupt service routine, you should use the standard RTI instruction—not an RTS. The RTI instruction transfers control back to the IM. (RTI is used because the IM saves additional status information in the processor status byte and then pushes this byte onto the stack.)

Automatic UNITCLEAR

Under version 1.2 of the Pascal system, devices are reinitialized when the system terminates abnormally. This can occur when a program gets a system error or when a user interrupts the program from the keyboard (CONTROL-@).

To reinitialize, the system executes a UNITCLEAR on *all* devices (1-20 and 128-143). This is done even when the driver's attach data record specifies that no initialization is to be done at startup time.

This presents a problem when the UNITCLEAR portion of a driver contains code to initialize the service routine chain (as described above). Drivers under version 1.2 must have some code to distinguish between the first initialization (which sets up the driver chain) and any subsequent initialization (call to UNITCLEAR).

In the driver, these two kinds of initialization can be distinguished by a simple check of a byte of memory to see which type of initialization code needs to be run (if any). This is the scheme used in the example in Appendix B.

The 1.2 Pascal system reinitializes all devices because some drivers may have pointers into the stack/heap space. If this space were released without reinitializing the device drivers, the pointers would then point to invalid code or data. The problem can't be solved by simply disabling further interrupts, since some external devices (for example, a remote network printer server) need to be notified of the reset; if interrupts were disabled, information coming back from the remote device could not be handled correctly.

Environmental Considerations

If your card uses the $C800 expansion space, then location $7F8 must contain the value $Cn, where *n* is the slot number of the card. The reason for this is that when you are executing in your $C800 space and an interrupt occurs, the interrupt service routine may decide to use its own $C800 space. Once the interrupt has been serviced, the system must know if it needs to reselect the $C800 space for your card. The IM will take the contents of location $7F8 (which can be initialized any time before your driver enters the $C800 space), and use this number to reselect your card.

If you do not do this, it is very possible that your routines may not work correctly since your $C800 space will not be reselected. The only other way to solve this is to disable all interrupts while you are in your $C800 space.

If your interrupt service routine does need to modify the contents of $7F8, it must save the contents and then restore the previous value before it is done.

It is *not* necessary to save registers in an interrupt service routine. The IM saves them before jumping to the chain of service routines, and restores them before resuming normal execution of the interrupted code.

There are additional restrictions on interrupts for applications that execute under the 64K Pascal system and that also use the auxiliary 64K memory on an Apple IIe. Since the IM and all interrupt service routines are resident in the main RAM, if an interrupt occurs while the application is using the auxiliary RAM, the interrupt will not be serviced properly and may cause the system to crash. For this reason, an application must disable interrupts while the auxiliary 64K is in use or must be able to handle the interrupt management itself. This does not apply to the Apple IIc.

On the Apple IIe, the IM will save the state of the 80STORE and PAGE2 soft switches, and will deselect PAGE2 if 80STORE is selected. The original state of the PAGE2 switch is restored after the interrupt is serviced. On the Apple IIc, this is done by the firmware rather than the IM.

In the 128K Pascal system on the Apple IIe, the IM will also save the state of the RAMRD and RAMWRT soft switches and will then select read main RAM and write main RAM. The original state is restored after the interrupt is serviced. On the Apple IIc, this is done by the firmware rather than the IM.

If an interrupt service routine uses any zero page user temporaries ($0-$35), then it must save their contents, and restore them after the interrupt has been serviced.

If an application switches in the Monitor ROM, it must disable interrupts before it does so. This does not apply to the Apple IIc.

Attaching an Interrupt-Based Driver

Attaching an interrupt-based device driver is no different from attaching any user-defined device driver: you must place the driver code in the ATTACH.DRIVERS library, execute ATTACHUD to create an ATTACH.DATA data record, and make sure that these files, plus SYSTEM.ATTACH, are on your startup disk when you start the Pascal system.

For more details, see Chapter 3. Here are some reminders that pertain to interrupt-based drivers:

When you run ATTACHUD, *be sure* to tell it that your driver uses interrupts. This ensures that a data record for the IM is present in your ATTACH.DATA file. Also remember to tell ATTACHUD that your driver must be initialized at startup time.

On the 64K Pascal system, you *must* include IM.CODE in ATTACH.DRIVERS. See the following section for details.

Installing the IM

The Interrupt Manager (IM) driver is shipped in the file IM.CODE on the *Attach Tools* disk.

On a 128K Pascal System

Interrupt handling is built into the 128K Pascal system, and you don't need to install IM.CODE. If IM.CODE is present in ATTACH.DRIVERS, it will simply be ignored.

For each interrupt-based driver that you attach, remember to tell ATTACHUD that it uses interrupts. When you do so, ATTACHUD ensures that a data record for the IM code is included in the ATTACH.DATA file. This data record is included in ATTACH.DATA as long as at least one of your drivers uses interrupts.

On a 64K Pascal System

If you are running a 64K Pascal system, you must use the LIBRARY utility (described in Chapter 8 of the *Apple Pascal Operating System Reference Manual* for the Apple II) to install IM.CODE into the ATTACH.DRIVERS file. When you use LIBRARY, the size of IM.CODE is shown as approximately 280 bytes, but much of this space is occupied by relocation code that is not resident at run time. At run time, the IM occupies about 200 bytes.

Although the IM is handled as if it were a driver, it cannot be called from a user program. It is automatically attached to the system at startup time.

For each interrupt-based driver that you attach, remember to tell ATTACHUD that it uses interrupts. When you do so, ATTACHUD ensures that a data record for the IM code is included in the ATTACH.DATA file. This data record is included in ATTACH.DATA as long as at least one of your drivers uses interrupts.

Chapter 5

Special BIOS Considerations

The Pascal system's BIOS (Basic Input/Output Subsystem) is a package of device drivers and other support routines.

The code for the BIOS resides in the bank-switched portion of the Pascal language card. The Pascal system calls BIOS routines through a pair of jump tables. When SYSTEM.ATTACH loads a user-defined device driver, it places a pointer to the new driver in another jump table. Then the driver can be called from the BIOS.

This chapter explains this bank-switching scheme in detail. You will need this information when you write a user-defined device driver if your driver code is to call BIOS routines. This chapter also contains information on special characters (which are usually handled by RSP routines) and managing the type-ahead buffer.

Calling BIOS Routines

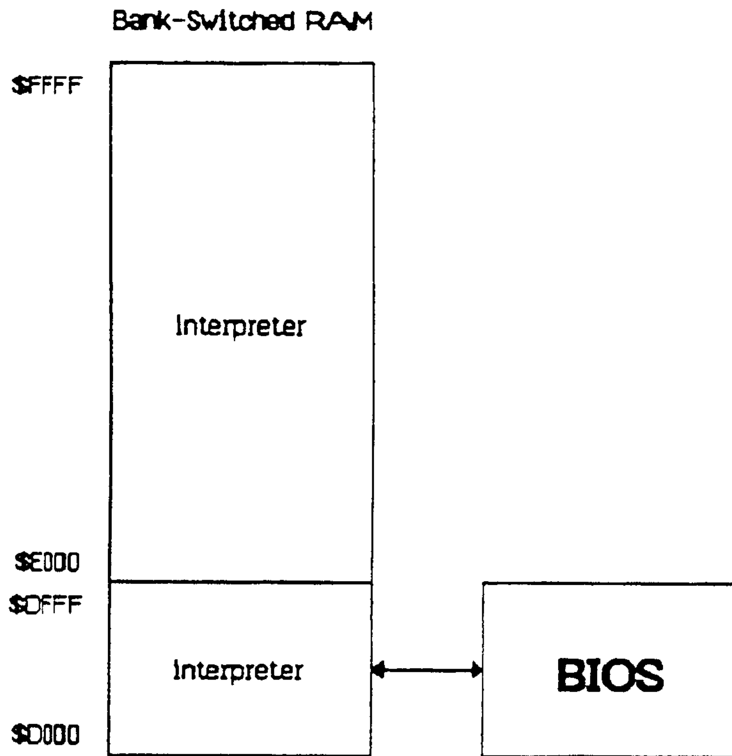
BIOS routines are listed and described in Table 5-1. This section explains in detail how these routines are called.

RAM addresses \$D000 to \$DFFF on the Apple IIe and IIc and the Apple II language card can be switched between two separate banks of memory. Under the Apple II Pascal system, one of these banks contains a portion of the system's interpreter. The other bank contains the BIOS.

The RSP calls a BIOS routine by way of a jump table in the interpreter. A routine called via this jump table folds into main memory the bank that contains the BIOS. When the BIOS routine has been executed, the interpreter bank is folded back in, and control returns to the Pascal interpreter.

Figure 5-1 illustrates this bank-switching scheme.

Figure 5-1. *Bank Switching for the BIOS*



Further description of bank switching can be found in the *Apple Language Card Installation and Operation Manual* for the Apple II and the *Apple IIe Reference Manual.*

The Interpreter's Jump Table

The interpreter's jump table to the BIOS, labeled simply BIOS, is called by the RSP. Each of the entries in the BIOS jump table calls a routine named SAVERET. SAVERET stores some return addresses for a given routine and then folds in the BIOS bank.

Here is the jump table called BIOS:

```
BIOS     JSR    SAVERET    ;Call CREAD
         JSR    SAVERET    ;Call CWRITE
         JSR    SAVERET    ;Call CINIT
         JSR    SAVERET    ;Call PWRITE
         JSR    SAVERET    ;Call PINIT
         JSR    SAVERET    ;Call DWRITE
         JSR    SAVERET    ;Call DREAD
         JSR    SAVERET    ;Call DINIT
         JSR    SAVERET    ;Call RREAD
         JSR    SAVERET    ;Call RWRITE
         JSR    SAVERET    ;Call RINIT
         JSR    SAVERET    ;Call IORTS (do nothing)
         JSR    SAVERET    ;Call GRINIT
         JSR    SAVERET    ;Call IORTS (do nothing)
         JSR    SAVERET    ;Call CSTAT
         JSR    SAVERET    ;Call ZEROSTAT
         JSR    SAVERET    ;Call DSTATT
         JSR    SAVERET    ;Call ZEROSTAT
KCONCK   JSR    SAVERET    ;Call CONCK
         JSR    SAVERET    ;Call UDRWI
         JSR    SAVERET    ;Call PSUBDRV
```

The offset to the actual BIOS routine is on top of the stack when the BIOS jump table is called. After the routine SAVERET has folded in the BIOS, it calls the routine SAVERET2, which does the actual jump, using the offset to the BIOS routine plus the offset to the second jump table, which is called BIOSAF.

Here is the code for SAVERET:

```
SAVERET   STA     TT1          ;Save the AREG (this is the
                               ;local return address)
          PLA                  ;Get the offset to the
                               ;desired routine
          CLC
          ADC     #05C-2       ;Add the offset to the
                               ;BIOSAF jump table
          STA     TT2          ;Save this in TT2
          PLA
          ADC #0
          STA TT3
          PLA                  ;Save the 2-byte Pascal
          STA     RETL         ;return addr in RETL-RETH
          PLA
          STA     RETH
          .IF     SYSSIZE<>48.  ;If the BIOS is not
                               ;already switched in
          LDA     0C083        ;...then fold it in
          .ENDC
          LDA     TT1          ;Restore the local return
                               ;address
          JSR     SAVERET2

          ...

SAVRET2   JMP     @TT2         ;Jump through the BIOSAF
                               ;jump table
```

BIOS Jump Tables

The BIOSAF jump table contains the addresses of all BIOS routines, in a defined order. SYSTEM.ATTACH stores a copy of the BIOSAF jump table on the heap, and stores the pointer to this table at $00E2. Your driver can use this copy of the table to get the address of a BIOS routine.

Each entry in BIOSAF consists of a JMP instruction followed by a two-byte address. Table 5-1 shows the order of the routines included in the table, along with a description of the routine and the offset to the JMP byte.

Table 5-1. *Order of Addresses in BIOSAF*

Routine Name	Description	Offset
CREAD	Reads from CONSOLE:	0
CWRITE	Writes to CONSOLE:	3
CINIT	Initializes CONSOLE:	6
PWRITE	Writes to PRINTER:	9
PINIT	Initializes PRINTER:	12
DWRITE	Writes to disk	15
DREAD	Reads from disk	18
DINIT	Initializes disk	21
RREAD	Reads from REMIN:	24
RWRITE	Writes to REMOUT:	27
RINIT	Initializes REMIN: and REMOUT:	30
IORTS	(Graph write—not implemented)	33
GRINIT	(Graph init—sets text mode)	36
IORTS	(Printer read—not implemented)	39
CSTAT	Status of CONSOLE:	42
ZEROSTAT	(Printer status—not implemented)	45
DSTATT	Status of disk	48
ZEROSTAT	(Remote status—not implemented)	51
CONCK	Checks CONSOLE: for characters typed	54
UDRWI	Routine to get user-defined driver	57
PSUBDRV	Routine to get substitute driver	60

The routine ZEROSTAT is a stub for status routines that have not been implemented. It simply pops parameters off the stack, stores 0 in the first word of the buffer (the ARRAY parameter in a UNITSTATUS call from Pascal), clears the XREG, and then returns. IORTS is a stub for routines other than status routines: it pops parameters, clears the XREG, and then returns.

The UDRWI routine calls a user-defined device driver by jumping through a jump table called UDJMPVEC. UDJMPVEC is declared as follows:

```
UDJMPVEC  JMP   0          ;Unit number 128 (usually
                           ;the ProFile driver)
          JMP   0          ;Unit number 129 (usually
                           ;the Mouse driver)
          JMP   0          ;Unit number 130 (available)

          ...

          JMP   0          ;Unit number 143 (available)
```

The actual driver addresses are filled in by SYSTEM.ATTACH at startup time. An address of 0 (the default value) indicates an unattached device.

The PSUBDRV routine is used to call user-defined drivers attached to device numbers 1-20. This range includes user-defined block-structured devices and user-defined drivers that substitute for standard Pascal system drivers (see Appendix E). Just as UDRWI uses the table UDJMPVEC, PSUBDRV uses a table called DISKNUM.

The jump table DISKNUM is declared as follows:

```
DISKNUM    .WORD  OFFFF       ;Unit #1  CONSOLE:
           .WORD  OFFFF       ;      2  SYSTERM:
           .WORD  OFFFF       ;      3  (see Appendix E)
           .WORD  0           ;      4  startup drive
           .WORD  1           ;      5  2nd disk drive
           .WORD  OFFFF       ;      6  PRINTER:
           .WORD  OFFFF       ;      7  REMIN:
           .WORD  OFFFF       ;      8  REMOUT:
           .WORD  4           ;      9  5th disk drive
           .WORD  5           ;     10  6th disk drive
           .WORD  2           ;     11  3rd disk drive
           .WORD  3           ;     12  4th disk drive
           .WORD  OFFFF       ;     13  a block-structured
                              ;         user device
           .WORD  OFFFF       ;     14  a block-structured
                              ;         user device

           ...                      ..  ...

           .WORD  OFFFF       ;     20  a block-structured
                              ;         user device
```

As with UDJMPVEC, the address of any user-defined driver is stored by SYSTEM.ATTACH. Note that in DISKNUM, this value will be the driver's address *minus 1.*

In DISKNUM, the status of each device is indicated by the high byte of the address word:

- If high byte = FF, then the device is not a disk drive (or it has not been attached).
- If high byte = 0, then the device is a standard disk drive, and the low byte contains its drive number.
- If high byte = anything else, then the word has been stored by SYSTEM.ATTACH, and contains the driver's address minus 1.

Note that since PSUBDRV calls a device driver via an RTS instruction, the address stored in the DISKNUM table must be 1 less than the driver's actual address (that is, .PROC-1). SYSTEM.ATTACH sets this up correctly.

If your driver is to call another BIOS routine, it must use the copy of BIOSAF that SYSTEM.ATTACH stores on the heap. The address of this copy is stored in ACJVAFLD at $0E2-E3. Your driver should jump through this address, using an offset to the appropriate driver. Note that this offset must be the number shown in Table 5-1 *plus 1*, to avoid the JMP instruction that is stored with each entry. The sample device driver in Appendix A uses this technique to call CONCK.

Code to call the CONCK routine is also stored in a four-byte vector located at address $BF0A. The usual way for a program to call CONCK is to do a JSR BF0A. However, this must *not* be done from within a user-defined device driver, as the return address to Pascal will be lost.

Returning to the Pascal Interpreter

The routine called GOBACK accomplishes the return to the Pascal interpreter:

```
GOBACK    STA    TT1          ;Save the AREG
          LDA    RETH         ;Get the Pascal return addr
          PHA                 ;in RETL-RETH, and push it
          LDA    RETL         ;on the stack
          PHA
          .IF    SYSSIZE<>48. ;If the BIOS is in main
                              ;memory
          LDA    0C08B        ;...fold in the interpreter
          .ENDC
          LDA    TT1          ;Restore the AREG
          RTS                 ;Return to Pascal
```

Handling Special Characters

Three types of special characters are handled by the Pascal system:

- *Type A* Characters used to control the 40-column display: CONTROL-A and CONTROL-Z (also CONTROL-W and CONTROL-E on an Apple II or Apple II Plus).
- *Type B* Characters to control output to the console: the START/STOP character, the FLUSH character, and the BREAK character.
- *Type C* Characters handled by the RSP: CONTROL-C, DLE (for blank compression), and CR (an LF may be inserted after each CR).

Characters of Type A or B are processed by the BIOS routine CONCK. This special handling can be turned off for the duration of a call to UNITWRITE. This is done by setting bits in the MODE parameter. (See Figure 2-1 for a diagram of the MODE parameter.)

The RSP inspects the MODE parameter and sets bits in a byte called SPCHAR according to the MODE parameter's value. When CONCK is called, it inspects SPCHAR, and processes (or does not process) special characters according to this scheme:

- If bit 4 in MODE is set, then bit 0 in SPCHAR is set—and Type A characters (40-column control) are not handled.
- If bit 5 in MODE is set, then bit 1 in SPCHAR is set—and Type B characters (STOP, FLUSH, and BREAK) are not handled.

If your program directly sets bits 0 or 1 in SPCHAR, then the associated special character checking will be turned off until either you reset the bits or the Pascal system is restarted. When special character checking is turned off, characters of Type A and Type B are passed back to the Pascal level without modification. You can use this feature if you wish to use some Type A or Type B characters for a special purpose in a particular application program, or if you simply wish to disable special character handling.

Redefining the Value of Type B Characters: Default values for the three Type B special characters are CONTROL-S for START/STOP, CONTROL-F for FLUSH, and CONTROL-@ for BREAK. You can redefine the value of these characters using the system reconfiguration utility. This utility saves a variety of information in the file SYSTEM.MISCINFO. The contents of SYSTEM.MISCINFO are in turn loaded into the Pascal system's SYSCOM buffer at startup time. For more information, see Appendix E or the section entitled System Reconfiguration in Chapter 8 of the *Apple Pascal Operating System Reference Manual* for the Apple II.

Special characters of Type C (CONTROL-C, DLE, and CR) are handled directly by the RSP.

If bit 2 (NOSPEC) of the MODE parameter equals 1, this disables the expansion of DLE blank compression codes. If bit 3 (NOCRLF) of the MODE parameter equals 1, this disables CR-LF expansion. If the MODE parameter is not included in a Pascal call to UNITREAD or UNITWRITE, it is passed as its default value of 0 (no bits set).

The CONTROL-C character (ASCII ETX) is used in the Pascal system to indicate the end of a file (except in the Editor, where CONTROL-C receives special treatment). If the Pascal Filer reads a CONTROL-C, it quits and returns to the operating system, which is reinitialized. If the Pascal operating system reads a CONTROL-C, it is reinitialized.

Managing the Type-Ahead Buffer

The type-ahead buffer saves input from the CONSOLE: device. It is managed by the CONCK routine.

Generally, the type-ahead buffer is read only by Pascal system programs. It isn't a good idea for your programs to take characters from this buffer. It is conceivable, though, that you might want to place characters from some device other than CONSOLE: into this buffer so that they will be read by the system.

The type-ahead buffer has room for 64 characters ($40 hex). The first character in the buffer is located at $03B1.

To manage the buffer, CONCK uses two pointers: a read pointer, RPTR (located at $BF18), and a write pointer, WPTR (located at $BF19). Whenever CONSOLE: is initialized, WPTR is set equal to RPTR.

When CONCK detects a character from CONSOLE:, it compares WPTR to $40. If WPTR equals $40, it is reset to $0 (note that this means the buffer is circular). CONCK then compares WPTR to RPTR. If the values are equal, the buffer is full and the character is ignored. If the buffer is not full, then the character is stored at ($03B1 + the value in WPTR).

To read from the type-ahead buffer, compare RPTR to WPTR. If the values are equal, the buffer is empty and the program must loop on a call to CONCK until a character is typed. If the values are not equal, RPTR is incremented and then compared to $40. If RPTR equals $40, it is reset to $0. The character to be read is now at ($03B1 + the value in RPTR).

Permanent Locations Used by the BIOS and Interpreter

The BIOS uses variables stored in three permanent locations: the zero page, the $BF00 page, and the $FF00 page within the $F800 bank that contains interpreter values and jump tables. The tables below show the most significant of these locations, including locations mentioned in the preceding sections of this chapter.

Warning
The tables in this section are meant to show certain important memory locations. They are incomplete. You must not assume that if a memory location is not shown below, it is not used by the BIOS or the interpreter.

Table 5-2. *Zero Page Locations*

Name	Location	Description
ACJVAFLD	0E2	Pointer to heap copy of BIOSAF
	0E3	(set up by SYSTEM.ATTACH).
RTPTR	0E4	Pointer to READTBL
	0E5	(see Appendix E)
WTPTR	0E6	Pointer to WRITTBL
	0E7	(see Appendix E)
UDJVP	0E8	Pointer to UDJMPVEC table
	0E9	
DISKNUMP	0EA	Pointer to DISKNUM table
	0EB	
JVBFOLD	0EC	Pointer to jump table before fold
	0ED	(that is, to BIOS in the interpreter)
JVAFOLD	0EE	Pointer to jump table after fold
	0EF	(that is, to BIOSAF in the BIOS)
BAS1L	0F0	Pointer to screen 1: low byte
BAS1H	0F1	high byte
BAS2L	0F2	Pointer to screen 2: low byte
BAS2H	0F3	high byte
CH	0F4	Horizontal cursor location, 0...79
CV	0F5	Vertical cursor location, 0...23
FTEMP1	0F6	
FTEMP2	0F7	
FSYSCOM	0F8	Pointer to SYSCOM
	0F9	(Pascal system communication area)
CONFLGS	0FA	Used to flag Type B characters

Table 5-3. *$BF00 Page Locations*

Name	Location	Description
CONCKVECTOR	0BF0A	Vector to call CONCK
	0BF0B	(do not use in a user-defined driver)
	0BF0C	
	0BF0D	
SCRMODE	0BF0E	Screen mode byte
LFFLAG	0BF0F	Line feed flag
EORCHAR	0BF11	
CURSFLAG	0BF12	
RANDL	0BF13	Random number seed: low byte
RANDH	0BF14	high byte
KEYCOUNT	0BF15	
BREAK	0BF16	BREAK vector
	0BF17	
RPTR	0BF18	Read pointer for type-ahead buffer, $0...40
WPTR	0BF19	Write pointer for type-ahead buffer, $0...40
RETL	0BF1A	Addr for return from BIOS call: low byte
RETH	0BF1B	high byte
SPCHAR	0BF1C	Status of Type A and B character handling
IBREAK	0BF1D	Address of BREAK
	0BF1E	
ISYSCOM	0BF1F	Address of SYSCOM
	0BF20	
		(IBREAK and ISYSCOM are used by CONINIT; see Appendix E)
VERSION	0BF21	Version number: 0 = Pascal 1.0 2 = Pascal 1.1 3 = Pascal 1.2
FLAVOR	0BF22	"Flavor" of system—development, runtime, etc. (see the *Pascal 1.2 Update*)
SLTTYPS	0BF27	Table of I/O card types (see Appendix D)
	0BF28	
	0BF29	
	0BF2A	
	0BF2B	
	0BF2C	
	0BF2D	
	0BF2E	
XITLOC	0BF2F	
	0BF30	
IIEFLAG	0BF31	(See the *Pascal 1.2 Update*)

Table 5-4. *$FF00 Page Locations*

Name	Location	Description
(no name)	FFF6 FFF7	Version word (for internal use)
(no name)	FFF8 FFF9	The start vector
(no name)	FFFA FFFB	Nonmaskable interrupt vector (NMI)
(no name)	FFFC FFFD	RESET vector
(no name)	FFFE FFFF	Interrupt request vector (IRQ)

Appendix A

Sample Code for a Device Driver

```
                ;Locations 0..35 hex may be used as pure
                ;temps. You should never assume the data
                ;in these locations will be safe if you
                ;leave the environment of the driver itself.
                ;Leaving includes calls to CONCK.

CONCKAD         .EQU 2

                ;Only one .PROC may occur in a driver. Each
                ;driver to be attached must be assembled
                ;separately using the Pascal Assembler, and
                ;can have no external references.

                .PROC   U128DR

                STA     TEMP1           ;Save AREG contents (dev #)
                PLA
                STA     RETURN
                PLA
                STA     RETURN+1
                TXA                     ;Use the XREG to tell you
                                        ;what kind of call this is
                                        ;(INIT, READ, etc.)
                CMP     #2
                BEQ     INIT
                CMP     #4
                BEQ     STATUS
                CMP     #0
                BEQ     PMS
                CMP     #1
                BEQ     PMS

                ;In case the XREG isn't valid ...
                JMP     RET
```

```
PMS     PLA                     ;Get the parameters off the
                                ;stack
        STA     BLKNUM          ;BLOCKNUMBER
        PLA
        STA     BLKNUM+1
        PLA
        STA     BYTECNT         ;LENGTH
        PLA
        STA     BYTECNT+1
        PLA
        STA     BUFADR          ;Address of ARRAY
        PLA
        STA     BUFADR+1
        PLA
        STA     UNITNUM         ;Also saved in TEMP1
        PLA
        STA     UNITNUM+1       ;Should always be 0
        PLA
        STA     CONTROL         ;MODE
        PLA
        STA     CONTROL+1
        TXA
        BNE     WRITE

READ    JSR     GOTOCK
        ;Your driver's code for a read goes HERE
        ;(If your driver handles more than one
        ;device, this code could jump to various
        ;places depending on the device # saved
        ;in TEMP1)
        JMP     RET

WRITE   JSR     GOTOCK
        ;Your driver's code for a write goes HERE
        JMP     RET

        ;If you wanted to call CONCK whenever your
        ;device did a read or write, you would
        ;use this routine:
CKR     .WORD   CONCKRT-1
GOTOCK  LDY     #55.            ;Offset to address of CONCK
        LDA     @OE2,Y
        STA     CONCKAD
        INY
        LDA     OE2,Y
        STA     CONCKAD+1
```

```
         LDA     CKR+1          ;Set it up so you return to
         PHA                    ;CONCKRT after the CONCK call
         LDA     CKR
         PHA
         JMP     @CONCKAD       ;Jump to CONCK

CONCKRT  RTS                    ;Return to caller

INIT     ;Your driver's code for initializing goes
         ;HERE
         JMP     RET

STATUS   PLA                    ;Get parameters
         STA     BUFADR         ;Address of status record
                                ;(PAB)
         PLA
         STA     BUFADR+1
         PLA
         STA     CONTROL        ;Control parameter
         PLA
         STA     CONTROL+1
         ;Your driver's code for status information
         ;goes HERE

RET      LDA     RETURN+1
         PHA
         LDA     RETURN
         PHA
         LDA     TEMP1
         RTS

RETURN   .WORD   0              ;Can't use 0 page for these
TEMP1    .WORD   0              ;since we leave our
                                ;environment when going to
                                ;CONCK

CONTROL  .WORD   0
UNITNUM  .WORD   0
BUFADR   .WORD   0
BYTECNT  .WORD   0
BLKNUM   .WORD   0

         .END
```

Appendix B

Sample Code for an Interrupt-Based Driver

```
;This sample driver is a user-defined device
;driver. It shows both the overall skeleton
;of a user driver and more importantly it
;shows how to write an interrupt-based device
;driver that uses the interrupt manager (IM).

;Macro subroutines

;Save/restore word off the stack (used to
;save return addresses)
.MACRO POP
PLA
STA     %1
PLA
STA     %1+1
.ENDM

.MACRO PUSH
LDA     %1+1
PHA
LDA     %1
PHA
.ENDM
;The ol' switch macro (see Chapter 6 of the
;Apple III Pascal Technical Reference Manual
;for description)
.MACRO SWITCH
.IF     "%1" <> ""
LDA     %1
```

```
                .ENDC
                .IF     "%2" <> ""
                CMP     #%2+1
                BCS     $010
                .ENDC
                ASL     A
                TAY
                LDA     %3+1,Y
                PHA
                LDA     %3,Y
                PHA
                .IF     "%4" <> "*"
                RTS
                .ENDC
$010            .ENDM
                ;Move first word into the second
                .MACRO MOVE
                LDA     %1
                STA     %2
                LDA     %1+1
                STA     %2+1
                .ENDM


                ;Equates

                ;Zero page (0-$35 is available) is used for
                ;return addresses and global temps

                ;Zero page temporary locations
CSLIST          .EQU    0               ;Buffer address
CTRLWORD        .EQU    2               ;Storage for control word

IRQ             .EQU    0FFFE           ;IRQ vector location
FLAG6522        .EQU    0C2ED           ;Interrupt flag register for
                                        ;hypothetical card in slot 2

                ;Error code equates

                ;Upon completion of the driver, the X
                ;Register will hold an appropriate error
                ;code that will be converted into the Pascal
                ;reserved variable IORESULT. The Pascal
```

```
        ;program should check IORESULT after all
        ;UNITSTATUS calls made to the driver. Error
        ;code numbers 128-255 are to be used by your
        ;driver.

XNOERRS  .EQU   0           ;No errors encountered
XBADCMD  .EQU   3           ;Bad command to driver
ERRCODE  .EQU   128.        ;User-defined error message

         .PROC  SAMPLE

        ;This is the main entry point for the ATTACH
        ;driver. This driver is defined as a user
        ;device and therefore will only be used from
        ;Pascal using direct I/O (i.e. UNITSTATUS,
        ;UNITREAD/WRITE).

        ;Upon entrance, the X Register will contain
        ;the type of call requested (UNITREAD, WRITE,
        ;CLEAR, etc.). See Chapter 2 for more
        ;details on the stack setup.

START
         POP    RETURN      ;Save return address
         TXA                ;Get type of call
         SWITCH ,4,IDTABLE

BADREQ   LDX    #XBADCMD    ;If you got here, the call
                            ;is in error!
         BNE    GOBACK      ;Always taken
GOBACKOK LDX    #XNOERRS    ;Go here if you want to
                            ;return with no errors
GOBACK   PUSH   RETURN      ;Or return with X Register
                            ;holding error code
         RTS                ;Main exit point

IDTABLE  .EQU   *
         .WORD  READ-1
         .WORD  WRITE-1
         .WORD  INIT-1
         .WORD  BADREQ-1
         .WORD  U_STATUS-1
```

```
            ;INIT does two things: 1) moves the IRQ
            ;vectors to the appropriate locations the
            ;first time called and 2) every additional
            ;call will be meant to issue an appropriate
            ;initialization request to the driver (if
            ;required).
INIT
            PHP
            SEI                     ;Disable interrupts
            LDA     TYPE
            BEQ     $001            ;If zero then do init stuff
            .
            .
            Any UNITCLEAR call after the initial one by
            the system will jump to this area
            .
            .
            JMP     $090            ;Always taken
$001        INC     TYPE            ;Bump type field so next
                                    ;time we don't do it

            ;This next section moves the IRQ vector into
            ;a temporary location. The MOVE macro is a
            ;16-bit move instruction. See above macros
            ;for an explanation.

            MOVE    IRQ,JUMPTO
            MOVE    INTADR,IRQ ;Patch IRQ location and jump
                                    ;vector
            .
            .
            More code for initial initialization call
            .
            .
$090        PLP
            JMP     GOBACKOK
INTADR      .WORD   INTHNDLR        ;Interrupt handler address
JUMPTO      .WORD   0               ;Save area for next
                                    ;interrupt svc routine
TYPE        .BYTE   0               ;If zero then init call;
                                    ;else cleanup call
RETURN      .WORD   0               ;Return address for Pascal
```

```
            ;READ is called when the program generates a
            ;UNITREAD
READ
            .
            .
            Code for the UNITREAD call
            .
            .
            LDX     #ERRCODE   ;Error completion code
            JMP     GOBACK

            ;WRITE is called when the program generates a
            ;UNITWRITE
WRITE
            .
            .
            Code for the UNITWRITE call
            .
            .
            LDX     #ERRCODE   ;Error completion code
            JMP     GOBACK

            ;U_STATUS is called when the program executes
            ;a UNITSTATUS call to this particular device.
            ;
            ;The order of the stack (four bytes) is:
            ;
            ;TOS =>POINTER TO STATUS RECORD
            ;
            ;CONTROL WORD
            ;   bits  15..13    12..2        1        0
            ;          user    reserved   status/    dir.
            ;        defined   for future control
            ;
            ;   direction - 0 = status of output channel
            ;               1 = status of input channel
            ; status/ctrl - 0 = status call
            ;               1 = control call
            ;
            ; Bits 13-15 should have the number of the
            ; control/status request.
```

```
        ;From Pascal, the call should be:
        ;
        ;   UNITSTATUS(130,BUFFER,OPTION)
        ;
        ;where:
        ;
        ;  130 = Device driver number (currently
        ;    130 is used by this driver)
        ;  BUFFER = PACKED ARRAY [0..??] OF 0..255;
        ;     This array is as big as needed
        ;     by the code called.
        ;  OPTION = PACKED RECORD
        ;     DIRECTION : 0..1;
        ;     STAT_CTRL : 0..1;
        ;     RESERVED : 0..2047;
        ;     CODE : 0..7;
        ;   END;
U_STATUS
        POP     CSLIST      ;See above
        POP     CTRLWORD    ;Ditto
        LDA     #02         ;Mask for status/control bit
        BIT     CTRLWORD    ;If bit 2 is set, zero flag
                            ;will be cleared
        BNE     CONTROL     ;Go do a control call
         JMP    STATUS      ;Status request

        ;This is the unitstatus call, control request
        ;section

CONTROL
        LDA     CTRLWORD+1 ;Get control word
        AND     #0E0        ;Is it #0   :  0000 0000
        BEQ     CODE_ZERO   ;Yes
        CMP     #20         ;Is it #1   :  0010 0000
        BEQ     CODE_ONE    ;Yes
        CMP     #40         ;Is it #2   :  0100 0000
        BEQ     CODE_TWO
        CMP     #60         ;Is it #3   :  0110 0000
        BEQ     CODE_THREE
        LDX     #XBADCODE   ;Completion code error
        JMP     GOBACK
```

```
CODE_ZERO
          .
          .
          Code for control code zero
          .
          .
          LDX     #ERRCODE   ;Completion code error
          JMP     GOBACK     ;And report it
CODE_ONE
          .
          .
          Code for control code one
          .
          .
          LDX     #ERRCODE   ;Completion code error
          JMP     GOBACK     ;And report it

          ;Repeat for codes 2 and 3

CODE_TWO
          .
          .
          LDX     #ERRCODE   ;Completion code error
          JMP     GOBACK     ;And report it
CODE_THREE
          .
          .
          LDX     #ERRCODE   ;Completion code error
          JMP     GOBACK     ;And report it

          ;This is the unitstatus call, status request
          ;section

STATUS
          LDA     CTRLWORD+1 ;Get control word
          AND     #OEO       ;Is it #0    :  0000 0000
          BEQ     SCODE_ONE  ;Yes
          CMP     #20        ;Is it #1    :  0010 0000
          BEQ     SCODE_TWO  ;Yes
          LDX     #XBADCODE
          JMP     GOBACK
```

```
SCODE_ONE
            .
            .
            Status code one
            .
            .
            LDX     #ERRCODE   ;Completion code error
            JMP     GOBACK     ;And report it
SCODE_TWO
            .
            .
            Status code two
            .
            .
            LDX     #ERRCODE   ;Completion code error
            JMP     GOBACK     ;And report it

            ;This is the interrupt handler. Remember that
            ;we don't have to save the context of the
            ;system. The code used to check for an
            ;interrupt will have to be changed depending
            ;on your hardware.
INTHNDLR
            ;First we check to see if we generated the
            ;interrupt

            LDA      FLAG6522 ;Check for 6522 int flag
            BPL     GOAHEAD    ;If bit was set, we
                               ;generated it
            JMP     NEXT       ;Otherwise go to the next
                               ;interrupt hndlr

            ;Since we generated the interrupt, we service
            ;it and then return to the interrupt manager
            ;with an RTI
```

```
GOAHEAD
                .
                .
                Interrupt handler code for our card
                .
                .
                RTI                     ;Go back to the interrupt
                                        ;manager

                ;If we got to NEXT, we must have decided that
                ;the interrupt was not generated by us.
NEXT
                JMP       @JUMPTO
                .END
```

Appendix C

Information Contained in ATTACH.DATA

This information is contained in an attach data record, created by running the program ATTACHUD:

- The name of the device driver.
- The device number(s) (unit numbers) attached to this driver.
- Whether to initialize the device(s) at startup time.
- Whether the driver's code is aligned. If so, to which byte boundary (0-256).
- Whether the driver can be placed in the first HiRes graphics page.
- Whether the driver can be placed in the second HiRes graphics page.
- Whether the driver uses interrupts.
- Whether the driver has transient initialization code. If it does, the name of the transient module. Whether the transient module's code is aligned. If so, to which byte boundary (0-256).

Under Apple II Pascal version 1.2, this information is saved in the format defined by the following declarations.

Warning
This record format is valid only for version 1.2. It is not the same as the version 1.1 format, and there is no guarantee that it will remain the same in future versions of Apple II Pascal.

```
alpha =     packed array [0..7] of char;

punits =    (undefu, consu, systu, graphu, dd4u, dd5u,
            printu, remiu, remou, dd9u, dd10u, dd11u,
            dd12u, dd13u, dd14u, dd15u, dd16u, dd17u,
            dd18u, dd19u, dd20u, ud128u, ud129u, ud130u,
            ud131u, ud132u, ud133u, ud134u, ud135u,
            ud136u, ud137u, ud138u, ud139u, ud140u,
            ud141u, ud142u, ud143u);

sopunits =  set of punits;

devrec =    packed record
               devname: alpha;
                  {Name of .PROC in the assembly code file
                  for this device.}
               uattached, initunits: sopunits;
                  {Units this device driver is to be attached
                  to. 1...20 for the system units and 21...36
                  for user device units 128...143.}
               drivalign: 0..256;
                  {This tells SYSTEM.ATTACH if the driver has
                  to be on some N byte boundary. N=0 =>
                  driver can be loaded anywhere (0 is
                  default). N=8 => driver must be loaded on
                  an 8 byte boundary (0, 8, 16, etc.). N=256
                  => driver must be loaded on a page
                  boundary.}
               ufhrp, ushrp: boolean;
                  {True if attach can use the associated
                  hires page to put its drivers into.
                  UFHRP is for the 2000..3FFF page.
                  USHRP is for the 4000..5FFF page.}
               has_interrupts: boolean;
                  {If the driver will use interrupts, this
                  will be TRUE.}
               transname: alpha;
                  {Name of the .PROC in the assembly code
                  file for the driver's transient
                  initialization code if it exists, else
                  spaces.}
```

```
    transalign: 0..256;
       {This tells SYSTEM.ATTACH if the transient
       has to be on some N byte boundary. See
       drivalign above. Only used if transname is
       nonblank.}
end;
```

Appendix D

Peripheral Card Firmware Protocol

This is a summary of the protocol for peripheral card firmware that is to communicate with programs in Apple II Pascal, version 1.2.

For more information, refer to the *Apple IIe Design Guidelines* manual.

Certain card slots are automatically assigned to Pascal devices. A firmware card in slot 1 is PRINTER: (unit #6), a firmware card in slot 2 is REMOTE: (unit #7 and #8, REMIN: and REMOUT:), and a firmware card in slot 3 is CONSOLE: and SYSTERM: (unit #1 and #2).

The I/O routine entry point branch table is located near the beginning of the $Cs00 address space (where s is the slot number where the peripheral card is installed).

Before the BIOS calls firmware routines, it disables the $C800 ROM space of all other firmware cards. Then it checks four firmware bytes to identify the peripheral card.

Both the identifying bytes and the branch table are near the beginning of the $Cs00 ROM space. The identifiers are the following:

Address	**Value**
$Cs05	(See the following table)
$Cs07	(See the following table)
$Cs0B	$01 (Generic indication of a firmware card)
$Cs0C	$ci (Device signature; see below)

At startup time, the system initializes the table called SLTTYPS (located at $0BF27-0BF2E; see Table 5-3). This table contains an entry for each of the eight card slots on the Apple II. The value of the entry indicates the type of peripheral card that the slot

contains, as follows:

Value in SLTTYPS	Meaning	Value in $Cs05	Value in $Cs07
0	No PROM on card	—-	—-
1	Card not recognized	—-	—-
2	Disk controller card	$03	$3C
3	Com card	$18	$38
4	Serial card	$38	$18
5	Printer card	$48	$48
6	Firmware card	$48	$48

The four high-order bits, c, of the device signature byte identify the device class. They are as follows:

Value of c	Class
$0	Reserved
$1	Printer
$2	Joystick or other X-Y input device
$3	Serial or parallel I/O card
$4	Modem
$5	Sound or speech device
$6	Clock
$7	Mass storage device
$8	80-column card
$9	Network or bus interface
$A	Special purpose (none of the above)
$B - $F	Reserved for future expansion

The four low-order bits, i, of the device signature byte are an identifier for the card assigned by the card's manufacturer. It is not guaranteed to be unique. For example, the Apple IIe 80-Column Text Card has a device signature of $88. Application programs can use the device signature to identify specific devices. The value of i is ignored by the BIOS.

After Pascal has identified the peripheral card, it uses these branch table locations:

Address	Contents
$Cs0D	Initialization routine offset (required)
$Cs0E	Read routine offset (required)
$Cs0F	Write routine offset (required)
$Cs10	Status routine offset (required)
$Cs11	$00 if control and interrupt routines are handled by the firmware

Notice that $Cs11 contains $00 only if the control and interrupt-handling routines are supported by the firmware.

Here are the entry point addresses, and the contents of the 6502 registers on entry to and exit from Pascal I/O routines:

Addr	Offset for	XREG	YREG	AREG
$Cs0D	Initialization			
	On entry:	$Cs	$s0	
	On exit:	error code	(unchanged)	(unchanged)
$Cs0E	Read			
	On entry:	$Cs	$s0	
	On exit:	error code	(unchanged)	character read
$Cs0F	Write			
	On entry:	$Cs	$s0	character to write
	On exit:	error code	(unchanged)	(unchanged)
$Cs10	Status			
	On entry:	$Cs	$s0	request (0 or 1)
	On exit:	error code	(changed)	(unchanged)

Request code 0 means "Are you ready to accept output?" Request code 1 means "Do you have input ready?" On exit, the reply to the status request is in the carry bit: carry clear means no, carry set means yes.

Appendix E

Drivers for Standard Devices

Although it is not a recommended practice, it is possible to write a device driver that replaces one of the standard Pascal system device drivers. Standard drivers have special requirements, which are described in this appendix.

The PSUBDRV routine in the BIOS enables the RSP to call a user-defined substitute for a standard driver. See Chapter 5 for a description of this mechanism.

Character Devices

The standard system character devices are unit #1 and #2 (CONSOLE: and SYSTERM:, which refer to the same device), unit #6 (PRINTER:), and unit #7 and #8 (REMIN: and REMOUT:, which also refer to the same device; we will call this device REMOTE: for convenience).

Character I/O is accomplished one character at a time. The character is placed in the A Register, and the BIOS routine is then called. The Y Register must contain the unit number, which can be referred to if a single driver is to access more than one device.

Note that this differs from a call to regular user-defined devices, where the A Register contains the unit number.

Also note that DLE codes, CR-LF expansion, and CONTROL-C are all handled by the RSP. The START/STOP, FLUSH, and BREAK characters are handled by CONCK. (These characters can be disabled by setting bits in SPCHAR, in which case they are not handled at all. See Chapter 5.)

The interpreter contains two tables that tell the RSP whether input and/or output is allowed for a particular character device. These tables are called READTBL and WRITTBL. They both have the same format, so we will use READTBL as an example.

```
READTBL              ;Table of routine addresses to be
                     ;used when making a READ call
                     ;to a standard device
                     ;Not used with block-structured devices
        .WORD  BIOS+CONREAD   ;CONSOLE:, unit #1
        .WORD  BIOS+CONREAD   ;SYSTERM:, unit #2
        .WORD  0              ;Unit #3
        .WORD  0              ;Unit #4, a disk
        .WORD  0              ;Unit #5, a disk
        .WORD  0              ;PRINTER:, unit #6, read only
        .WORD  BIOS+REMREAD   ;REMIN:, unit #7, remote
                              ;read only
        .WORD  0              ;REMOUT:, unit #8, remote
                              ;write only
```

In this table, the address called BIOS is the base address of the jump table before the BIOS itself has been folded in. CONREAD is the offset to the address of the CONSOLE: read routine (CREAD), and REMREAD is the offset to the address of the REMIN: read routine (RREAD).

If an entry in this table is 0, then the device is write-only.

WRITTBL has exactly the same format, but the nonzero entries correspond to standard write routines (for example, CWRITE for the CONSOLE:).

If your substitute for a standard device driver supports a write or read that is not normally allowed (for example, your printer can send characters as well as receive them), then you will have to modify READTBL or WRITTBL accordingly. This must be done at startup time.

The value of an entry that allows a read or write for a substitute driver must be the base address of the jump table before the fold (JVBFOLD) plus the offset to the PSUBDRV routine (this offset equals 60).

The pointer to READTBL is called RTPTR, and resides at $0E4-0E5. The pointer to WRITTBL is called WTPTR, and resides at $0E6-0E7.

The address of the BIOS jump table before the fold, is stored in JVBFOLD at $0EC-0ED.

A single driver can be attached to more than one of the system's character devices. If this is the case, the first thing the driver must do when it is called is check the PRINTER: and/or REMOTE: devices, and place any characters received into its own PRINTER: or REMOTE: queue.

After this, it must call CONCK in order to maintain the CONSOLE: buffer. Then the call can be handled in the usual way, except that its read routine must check the unit number (passed in the Y Register), and return the next character in the type-ahead buffer if YREG equals 1 or 2, the next character in the PRINTER: buffer if YREG equals 6, or the next character in the REMOTE: buffer if YREG equals 7 or 8.

CONSOLE: and SYSTERM:

The devices CONSOLE: (unit #1) and SYSTERM: (unit #2) must be driven by the same driver. Calls to CONSOLE: and SYSTERM: are equivalent from the driver's point of view: the RSP decides whether it should or should not echo a character that is typed.

Sample code for a CONSOLE: driver appears at the end of this appendix.

The CONCK routine will call the CONSOLE: driver (as well as vice versa). If your substitute driver has a routine that gets or processes characters, the first three bytes of the driver code must be a jump to this routine; otherwise, the first three bytes must be no-ops.

All other calls to your driver (calls through PSUBDRV) will jump to three bytes beyond the beginning of your driver's code. This should be the .PROC entry point.

For example, the beginning of your CONSOLE: driver's code might look like this:

```
JMP     MYCONCK    ;The first three bytes
.PROC   OWNCON     ;The normal entry point
STA     TEMP1
PLA
...                ;Et cetera
```

When the CONSOLE: must do a read or write, the only thing passed on the stack is the return address (at top of stack). The character to be read or written is in AREG.

When the CONSOLE: is initialized, the stack contains parameters as follows (these are all one word):

top of stack —>   return address
                    pointer to SYSCOM
                    pointer to BREAK
                    address

The BREAK and SYSCOM pointers must be stored in \$BF16-BF17 and \$00F8-00F9, respectively. Also, the FLUSH and START/STOP conditions must be reset, and the type-ahead buffer cleared. This means setting SPCHAR, RPTR, and WPTR to 0 (see Chapter 5).

When CONSOLE: receives a status call, the stack contains these parameters (all one word):

top of stack —>   return address
                    control word
                    pointer to status
                    record

The control word has the format described in Chapter 2 (see Figure 2-2). The number of characters in the type-ahead queue must be stored in the first word of the status record. This value is the absolute value of (WPTR - RPTR). If type-ahead is not being used, a request for the status of output should always set the status record word to 0, and a request for the status of input should set the status record word to 1 if a character is waiting to be read, or 0 if no character is pending.

If the CONSOLE: read routine receives a character whose value is greater than or equal to 128, this is a control character such as 🍏. The next character in the buffer will be the character that was typed at the same time as the control key(s), and if it was pressed along with 🍏, its high bit will also be set.

The control character byte can have the following values:

\$84	means	🍏 was pressed (bits 7 and 2 are set)
\$90		🍎 was pressed (bits 7 and 4 are set)
\$C0		🍏-SHIFT was pressed (bits 7 and 6 are set)

If you have written your own version of CONCK to support a keyboard that does not have these special key combinations, the new CONCK routine must ensure that the high bit is clear for every character that it reads.

If the device your driver controls does not handle lowercase characters, the CONSOLE: write routine must map lowercase to uppercase.

The CONSOLE: write routine must handle certain special characters as described here:

CR ($0D)	A carriage return must move the cursor to the beginning of the current line.
LF ($0A)	A line feed must move the cursor to the same column of the next line. If the cursor begins at the last line of the screen, the screen should be scrolled up one line.
BELL ($07)	If possible, a sound should be made when this character is received. If this isn't possible, ignore BELL.
SP ($20)	Place a space at the cursor's current position, writing over whatever is there. Move the cursor to the next column. If the cursor is in the last column of the screen, do not move it. If the cursor is in the last column of the screen's last line, do not move it and do not scroll the screen. (The end-of-line and end-of-screen actions are not possible on all displays, and strictly speaking the cursor's action under these conditions is undefined.)
NUL ($00)	Delay for the amount of time required to write one character, but do not write anything.
Any Printable Character	Write the character to the screen and move the cursor as defined for SP.

The Type B characters—START/STOP, FLUSH, and BREAK—are soft characters. The default values are CONTROL-S, CONTROL-F, and CONTROL-@, respectively, but the user can change their values when the system is configured, as mentioned in Chapter 5. The current values are stored in the SYSCOM area. The zero page location FSYSCOM (see Table 5-2) contains a pointer to SYSCOM, and the offsets for these characters are as follows:

FLUSH	$053
BREAK	$054
START/STOP	$055

PRINTER:

In a PRINTER: (unit #6) read, write, or initialize call, the stack contains only the return address (on top of stack).

A PRINTER: init call should cause the printer to do a carriage return followed by a line feed. The printer's buffer should be cleared. The init routine should not do a form feed.

A PRINTER: status call has the same parameters as CONSOLE: status. The first word of the status buffer should be set to the number of characters in the printer's buffer for the direction requested (that is, input or output). If this can't be determined, set the status word to 0.

If your printer is a write-only device, then a call to PRINTER: read should set the IORESULT (that is, the X Register) to the error code 3. If your printer can read characters, you will have to have changed the READTBL.

If your printer is one that must receive an entire (buffered) line at once, the PRINTER: driver's write routine is responsible for the buffering.

The PRINTER: write routine must handle certain special characters as described here:

CR ($0D)	Print the current line and return the carriage to the first column. Don't do an automatic line feed.
LF ($0A)	Advance to the next line. If a carriage return must be sent with each line feed, it is all right to do so.
FF ($0C)	Move the paper to the top of form and then do a carriage return. If your printer cannot move to top of form, simply do a carriage return followed by a line feed.

REMIN: and REMOUT:

REMIN: (unit #7) and REMOUT: (unit #8) refer to a single device and are handled by a single driver. In Pascal, REMIN: is used to read from the device, and REMOUT: is used to write to it. This appendix calls the device REMOTE: for convenience.

Most often, the REMOTE: device is an RS-232 serial line. This is used in many different applications. REMOTE: should transfer data without modifying it in any way. The default rate of transfer is 9600 baud.

The stack format of calls to REMOTE: is the same as for PRINTER:. REMOTE: status should behave in the same way as PRINTER: status. REMOTE: init should do whatever is necessary to make the device ready to send and receive characters.

It is best if the input to REMOTE: can be buffered in the same way as input to CONSOLE:, but this is not a requirement.

Special Use of Unit #3

Unit #3 is not defined to access any device, but in Apple II Pascal, a UNITCLEAR call to unit #3 sets the screen to text mode. The Pascal system uses this feature, so you should not assign unit #3 to any driver.

Block-Structured Devices

The standard system block-structured devices have numbers 4, 5, and 9-12.

The driver for a block-structured device is responsible for mapping Pascal system block numbers into physical track-and-sector addresses.

Unit #4 (drive #0) is required to be the startup disk for the Pascal system. Once the system has started up, you can substitute the driver for unit #4, but the substitute device *must* have the same volume name as the disk that contained the bootstrap, and *must* have a copy of the file SYSTEM.PASCAL. In addition, the copy of SYSTEM.PASCAL must be identical to the copy on the startup disk (it must be exactly the same version), and it must be in the same location on the substitute disk (it must start at the same block number).

When a disk driver is called, the unit number is in the A Register (as it is for user-defined character devices).

In calls to disk drivers, the stack is set up as described in Chapter 2.

A status call to a disk driver must store the following information in the status record:

word 1: The number of bytes buffered in the requested direction (input or output). Return 0 if this is unknown.

word 2: The number of bytes per sector.

word 3: The number of sectors per track.

word 4: The number of tracks per disk.

As you can see, the last three words returned by the status request are constant for the type of disk drive being accessed.

As mentioned in Chapter 2, it is always safe to transfer a full block with UNITWRITE, but a full block should be transferred by UNITREAD only if LENGTH is a multiple of 512. If it is necessary for your driver to always transfer a full block, then your driver's UNITREAD routine must buffer the block itself, and then transfer no more than LENGTH bytes into the read buffer (the ARRAY parameter in a call from Pascal).

Sample Code for a CONSOLE: Driver

The following code includes the actual BIOS source for the console's INIT, READ, and WRITE routines.

```
                ;Initialize CONSOLE:

CINIT           PLA
                STA     FTEMP1          ;Save return address
                PLA
                STA     FTEMP2
                PLA
                STA     FSYSCOM         ;Save pointer to syscom area
                PLA
                STA     FSYSCOM+1
                PLA
                STA     BREAK           ;Save break address
                PLA
                STA     BREAK+1
                LDA     FTEMP2
                PHA                     ;Restore return address
                LDA     FTEMP1
                PHA
```

```
          LDA     RPTR          ;Flush type-ahead buffer
          STA     WPTR
          LDA     CONFLGS
          AND     #03E
          STA     CONFLGS       ;Clear STOP, FLUSH,
                                ;AUTO-FOLLOW bits
          JSR     TAB3          ;No, horizontal shift full
                                ;left
          LDX     #0            ;Clear IORESULT
          STX     KEYCOUNT      ;Set # chars buffered = 0
          RTS                   ;And return


          ;Read from CONSOLE:
          ;Keyboard, com or serial card in slot 3

CREAD     LDY     #030          ;Slot 3
          LDA     SLTTYPS+3     ;What type of card?
          CMP     #4            ;Is it a serial card?
          BNE     CREAD2        ;No, continue
          JSR     RSER          ;Yes, read it
          AND     #7F           ;Mask off top bit
          RTS
CREAD2
          CMP     #3            ;Is it a com card?
          BEQ     CREAD4
          CMP     #6            ;Is it a firmware card?
          BEQ     CREAD4
          JSR     CREAD4        ;Read the character
          PHA                   ;Save the character
DONTRMOV  PLA
          LDX     #0            ;Zero IORESULT
DONTCURS  RTS

SHOWRMOV  LDA     CURSFLG       ;Cursor on?
          BNE     DONTCURS      ;No, return
          JMP     INVERT        ;Apple screen: show cursor
                                ;Note INVERT will complete
                                ;subroutine with RTS
                                ;This routine space
                                ;optimized to save 1 byte

CREAD4    JMP     REALC4


          ;Write to CONSOLE:
          ;Video screen, com or serial card in slot 3
```

```
CWRITE    JSR    CONCK        ;CONSOLE character
                              ;available?
          BIT    CONFLGS      ;Is FLUSH flag set?
          BVS    CLRIO        ;Yes, discard character and
                              ;return
          TAX                 ;Save character in X
          LDY    #030         ;Slot 3;010
          LDA    SLTTYPS+3    ;What kind of card?
          CMP    #3           ;Com card?
          BEQ    WCOM         ;Yes, write to com card slot
                              ;3
          CMP    #4           ;Serial card?
          BEQ    WSER         ;Yes, write to serial card
                              ;slot 3
          CMP    #6
          BEQ    WFIRM
          STX    FTEMP1       ;Send character to screen
          JSR    SHOWRMOV     ;Remove cursor if it's
                              ;turned on
          LDY    CH
          JSR    VOUT2        ;Do the business
          JSR    SHOWRMOV     ;Show cursor if it's
                              ;turned on
CLRIO     LDX    #0           ;Clear IORESULT
          RTS                 ;Return from VIDOUT

WFIRM     TXA
          PHA
          LDA    #0
          JSR    IOWAIT
          JSR    SER1
          LDY    #0F
METAVEC   JMP    FVEC1


          ;Bump ring-buffer index routine for CONSOLE

BUMP      INX                 ;Bump buffer pointer with
                              ;wraparound
          CPX    #CBUFLEN
          BEQ    ZERIO        ;More crude space
                              ;optimizations
          RTS


          ;CONSOLE read loop
```

```
CREAD3    JSR   ADJUST     ;Horizontal scroll if
                           ;necessary

REALC4    JSR   CHARTEST   ;Test character
          BEQ   CREAD3     ;Loop till something in
                           ;buffer
          DEC   KEYCOUNT   ;Update # of chars buffered
                           ;excl prefixes
          JSR   NEWBUMP    ;Get the char into the
                           ;accumulator
                           ;Note: hi bit will be set
                           ;only if key was prefixed
          BPL   ZERIO      ;C063 and/or OPEN and/or
                           ;SOLID APPLE code?
                           ;No, clean up and leave
          JSR   NEWBUMP    ;Get the corresponding char
                           ;into the accumulator
                           ;Note: don't need to
                           ;validate X since OPEN
                           ;and/or SOLID APPLE code
                           ;always followed by
                           ;corresponding character
ZERIO     LDX   #0         ;Clear IORESULT
          RTS              ;And return to Pascal

NEWBUMP   LDX   RPTR       ;Make X point to the
                           ;character to be read
          JSR   BUMP
          STX   RPTR       ;Bump read pointer
          LDA   CONBUF,X   ;Get character from buffer
          RTS

CHARTEST  JSR   CONCK      ;Service keyboard
          LDA   KEYCOUNT   ;Set up Z flag for keypress
                           ;function
          RTS
```

Appendix F

IORESULT Codes

Device driver subroutines must return one of these IORESULT codes in the X Register:

Code	Meaning
0	No error
1	Bad block, parity error
2	Bad device number
3	Illegal I/O request (such as attempting to UNITREAD from PRINTER:)
4	Data-com timeout
5	Volume went off-line
6	File lost in directory
7	Bad filename
8	No room on volume
9	Volume not found
10	File not found
11	Duplicate directory entry
12	File already open
13	File not open
14	Bad input format
15	(Not used)
16	Disk is write-protected
17	Illegal block number
18	Illegal buffer address
19	Must read a multiple of 512 bytes (this is a ProFile error)
20	Unknown ProFile error
64	Device error: failed to complete a read or write
128 - 255	(Available to user-defined devices)

Index

A

A Register 12
.ABSOLUTE directive 11, 23
ADMERG 31-32
ADMERG.CODE xii
ARRAY parameter 16, 17
attach data record 26
  contents 79
  format 80-81
ATTACH TOOLS disk xii, 25, 26, 48
ATTACH.DATA 8, 27, 79-81
ATTACH.DRIVERS 8
attaching a driver 7-9, 25-26
ATTACHUD 26-30
ATTACHUD.CODE xii

B

bank switching 51-52
Basic Input/Output Subsystem (BIOS) 3-4
$BF00 page locations 62
BIOS See Basic Input/Output Subsystem
BIOS memory locations 61-63
BIOS routines 55
  calling 51-57
BIOSAF See jump tables

blank-compression code 17-18, 59
block-structured devices 5
  standard 93-94
BLOCKNUMBER parameter 16, 17
blocks 5, 17
break bit 39, 40, 41
BRK instruction 39, 40
byte boundary, aligning on 29

C

character devices 5
  standard 87-93
CONCK 14, 55, 57, 58, 60
CONSOLE: 89-92
CONTROL parameter 19-21
CONTROL-C 59
CONVAD 33-34
CONVAD.CODE xii
CR-LF expansion 17, 59

D

device drivers, standard 4, 87-97
device numbers 6, 28
devices
  block-structured 5
  character 5
DIRECTION 20
Disk II 5
DISKNUM See jump tables

E

error messages at startup time 34-35
error numbers, IORESULT 13

F

$FF00 page locations 63
firmware cards 83
  protocol for 83-85

G

GOBACK routine 57

H

handling interrupts 37-42
HiRes pages 27

I

IM See Interrupt Manager
IM.CODE xii, 47-49
INIT 12
initialization 9, 15
  transient 9
initialize subroutine 15
interpreter, Pascal 51, 57
interrupt flag register 39
interrupt handling 37-42
Interrupt Manager (IM) 37, 38
  installing 48-49
interrupt service routine 37-46
  initializing 43
interrupt-based driver
  attaching 47
  writing 42-47
interrupts
  disabling 42-43
  enabling 42-43
  handling 37-42
  recognizing 44
IORESULT 13
  codes 13, 99
IORTS 55
IRQ vector 38, 40

J, K

JMP byte, offset to 54-55
JMP instruction 54
jump tables
  BIOSAF 54
  DISKNUM 56
  interpreter's (BIOS) 53
  UDJMPVEC 55

L

LENGTH parameter 16, 17
LIBRARY utility 25, 26
LIBRARY.CODE 25
line-feed character 17

M, N, O

memory locations 61-63
MODE parameter 17-19, 58, 59
Mouse 6, 41

P, Q

PAB parameter 19
parameters
  ARRAY 16, 17
  BLOCKNUMBER 16, 17
  CONTROL 19-21
  LENGTH 16, 17
  MODE 17-19, 58, 59
  PAB 19
  UNITNUMBER 12, 17
partial blocks, transferring 17
Pascal interpreter 51, 57
PRINTER: 92
.PROC directive 12, 23
ProFile 6
PSUBDRV routine 56, 57
PURPOSE 21

R

READ 12
read subroutine 16
relocatable code 11
REMIN: 92-93
REMOUT: 92-93
RSP See Runtime Support Package
RTI instruction 42, 44
runtime support 3-4

Runtime Support Package
(RSP) 3-4, 51, 53
Runtime Systems xii, 37

S

SAVERET routine 53-54
sectors 5
SHOWAD 32
SHOWAD.CODE xii
6502 Assembler 11
soft switches 46
special characters 58-59
  Type A 58
  Type B 58, 59
  Type C 58, 59
standard device drivers 4,
87-97
standard devices 3, 21
startup disk 8, 25, 35
startup time
  error messages at 34-35
  initializing at 15
STATUS 12
status subroutine 19
subroutines, device driver
15-21
  initialize 15
  read 16
  status 19
  write 17
SYSCOM buffer 59
SYSTEM.ATTACH xii, 8, 9, 15,
34, 35
SYSTERM: 89-92

T

tracks 5
transient initialization
  code 29, 30
  modules 22-23, 42, 43
Type A characters 58
Type B characters 58, 59

Type C characters 58, 59
type-ahead buffer 14, 41, 60

U, V

UDJMPVEC See jump tables
UDRWI routine 55
unit #3 93
unit I/O procedures 5
unit numbers See device numbers
UNITCLEAR 5, 7, 9, 12, 15
  automatic 45
UNITNUMBER parameter 12, 17
UNITREAD 5, 7, 12, 15, 16-19
UNITSTATUS 5, 7, 12, 15, 19-21
UNITWRITE 5, 7, 12, 15, 16-19

W

WRITE 12
write subroutine 17

X, Y

X Register 12
  completion code in 13
  values 12
XREG See X Register

Z

zero page locations 61
ZEROSTAT routine 55