The Software Implementation of Parallel Device Handlers and Drivers, 1984-05-22 Version, Revision B

The document itself begins on the next page.

Document source:

Original backup tapes owned by Dutchman2000, obtained by Atarimania.
Documentary research and PDF layout by Laurent Delsarte.
Note that these backup tapes contain A LOT of information spread out in many folders, meaning it will take time to process the important bits.

Document identification:

Original file name:	BARALL.00006.DOC extracted from CEO.01JUN84
Title of document:	The Software Implementation of Parallel Device Handlers and Drivers, 1984-05-22 Version, Revision B
Author(s):	Rick K. Nordin (Richard K. (Hud) Nordin)
Original file date:	1984-05-22
Type of document:	Memo
Target audience:	Internal
Status:	Final
Reference (Atari):	(unknown)
Reference (Laurent Delsarte):	For any discussion, this PDF has been given the reference **BKUP-1984-05-22-MEMO-0011A-2** which should be quoted in any communication.
Tags:	#Atari #8bit #6502 #600XL #800XL #1450XLD #Parallel #PBI #Handler #Driver

Comments:

This version has been extensively revised compared with the previous one.
Note the mention of the 1450XLD, which is once again taken into consideration (previously 1250XLD in 1983).

In the “original version” (the “raw text version”), the author chose to express certain hexadecimal numbers with the postfix “H”, uppercase. Some, not all. For the sake of clarity, I've removed the “H” and added the prefix “$” in front of all hexadecimal numbers.

This page intentionally left blank

The Software Implementation of Parallel Device Handlers and Drivers, 1984-05-22 Version, Revision B

R. K. Nordin
Revision B
May 22, 1984

Table of Contents

1. Overview

Computers in the Atari 600XL / 800XL / 1450XLD line provide a parallel bus. Parallel devices on this bus physically include a ROM which contains code for handling I/O requests and driving the device.

This document describes the interface between the Operating System and a Parallel Device Handler and/or Driver.

As many as 8 parallel devices, numbered 0 through 7, may be on the parallel bus. A device's number is determined by hardware on the device, e.g., a configuration switch.

The ROM space in every parallel device is two kilobytes, addressed $D800 through $DFFF, in the same address space as the Operating System's Floating-Point Package.

When a parallel device is selected, addresses $D800 through $DFFF refer to locations within the device ROM. Selection of a device and its ROM is accomplished by setting the corresponding bit in hardware register $D1FF. For instance, storing $04 at $D1FF selects device 2. (It is illegal to set more than one bit in $D1FF.) Storing $00 at $D1FF deselects any parallel device and selects the Floating-Point Package.

Since the Floating-Point Package and each parallel device ROM occupy the same address space, the parallel device handlers and drivers may neither use the Floating-Point Package nor invoke any other code which uses the Floating-Point Package.

The Operating System is responsible for selecting the parallel devices at appropriate times, transferring control to the devices via fixed vectors within the device ROM's, receiving control back from the devices, and re-instating the Floating-Point Package.

A parallel device may be selected for the following four reasons:
1. Initialization
2. Processing of a Device Handler request
3. Processing of a low-level Serial I/O (SIO) type request
4. Processing of a parallel device interrupt request (IRQ)

A parallel device ROM, then, would provide code to process each of these four occurrences and vectors to the appropriate portions of code within the ROM.

In support of parallel device handlers and drivers, the Operating System apportions 512 bytes of RAM, addressed $D600 through $D7FF, to the 8 possible devices.

2. Parallel device ROM requirements

The Operating System requires a data table which starts at the low address of a parallel device ROM. This data table affirms the existence of a ROM for the device selected and provides vectors to routines within the ROM. Device selection and transferring of control from the OS will not be performed correctly unless this data is correct.

The data table consists of mandatory and optional entries. Only the mandatory entries are required for correct operation. The optional entries describe the ROM and device and only suggestions as to their use are given here.

It should be noted that the Device Handler Vector Table ($D80D through $D81C) has the same format as other Operating System Resident Handler (e.g., Printer Handler) vector tables.

Parallel Device ROM	Mandatory?	Data Table
$D800 - $D801	Optional	ROM Checksum (low byte, high byte)
$D802	Optional	Revision Number
$D803	**Mandatory**	ID Number 1 Value = $80
$D804	Optional	Name or Type
$D805 - $D807	**Mandatory**	Low-level I/O Vector Value = JMP address (low byte, high byte)
$D808 - $D80A	**Mandatory**	IRQ Handler Vector Value = JMP address (low byte, high byte)
$D80B	**Mandatory**	ID Number 2 Value = $91
$D80C	Optional	Device Name Value = Device Name in ASCII
$D80D - $D80E	**Mandatory**	Device Handler Open Vector Value = address-1 (low byte, high byte)
$D80F - $D810	**Mandatory**	Device Handler Close Vector Value = address-1 (low byte, high byte)
$D811 - $D812	**Mandatory**	Device Handler Get-Byte Vector Value = address-1 (low byte, high byte)
$D813 - $D814	**Mandatory**	Device Handler Put-Byte Vector Value = address-1 (low byte, high byte)
$D815 - $D816	**Mandatory**	Device Handler Status Vector Value = address-1 (low byte, high byte)
$D817 - $D818	**Mandatory**	Device Handler Special Vector Value = address-1 (low byte, high byte)
$D819 - $D81B	**Mandatory**	Initialization Vector Value = JMP address (low byte, high byte)
$D81C	Optional	Not Used Value = 0

3. General device selection process

Seven registers are used in the parallel device selection process:

1. PDVS ($D1FF), Parallel device select hardware register
2. SHPDVS ($0248), Parallel device select shadow
3. PDVMSK ($0247), Parallel device mask
4. PDVI ($D1FF), External parallel device IRQ hardware register
5. PDIMSK ($0249), External parallel device IRQ mask
6. IPDVI ($D1CF), Internal parallel device IRQ hardware register
7. IPDIMK ($0254), Internal parallel device IRQ mask

Writing a value to the parallel device selection register, PDVS, causes a device or the Floating-Point Package to be selected. Setting bit *n* of PDVS causes parallel device *n* to be selected. Storing $00 into PDVS causes the Floating-Point Package to be selected.

RAM location SHPDVS is a shadow of the value written into the device selection register, PDVS. Whenever the Operating System writes a value into PDVS, the same value is stored in SHPDVS. This is necessary because the device selection value is not available by reading PDVS.

For instance, if code within a device ROM needs to know the device's number, it can inspect SHPDVS to determine which device is currently selected. (N.B.: When storing values into PDVS and SHPDVS, you MUST store first into SHPDVS and then into PDVS.)

The parallel device mask, PDVMSK, is used to control the selection process in all cases except for initialization and IRQ processing. By convention, if bit *n* of PDVMSK is set then device number *n* exists on the parallel bus. If bit *n* of PDVMSK is clear, then device *n* will not be selected. The Operating System, itself, does not set PDVMSK. It is the responsibility of each device's initialization routine to set the PDVMSK bit corresponding to the device's number.

The parallel device IRQ hardware registers, PDVI and IPDVI, indicate which of the parallel devices initiated an IRQ. These are read-only locations. Bit *n* of PDVI is set if external parallel device *n* initiated an IRQ, and bit *n* of IPDVI is set if internal parallel device *n* initiated an IRQ. (External parallel devices are devices that are physically located outside the computer and are connected via the Parallel I/O Port. Internal parallel devices are devices that are built into the computer, e.g., the disk drive on the 1450XLD.)

The parallel device IRQ masks, PDIMSK and IPDIMK, are used to determine which parallel devices are allowed to receive IRQ requests. Parallel device *n* will be selected to handle its IRQ only if bit *n* in PDIMSK or IPDIMK (whichever is appropriate) is set. The Operating System does not set PDIMSK or IPDIMK. It is the responsibility of each device's initialization routine to set the PDIMSK or IPDIMK bit corresponding to the device's number if the device driver is to handle IRQ's.

N.B.: Clearing a device's bit in PDIMSK or IPDIMK merely prevents the OS from recognizing IRQ's initiated by that device; it does not prevent the IRQ's from occurring.

Consequently, if a parallel device initiates an IRQ when its bit in PDIMSK or IPDIMK is clear, the Operating System will go into an infinite loop.

In order to prevent an infinite loop, the parallel device handler must set its bit in PDIMSK or IPDIMK before it enables its hardware, and it must disable its hardware before it clears its bit in PDIMSK or IPDIMK.

4. Device initialization interface

4.1 Operating System initialization

During cold-start and warm-start initialization, after initializing resident device handlers, and before attempting to initialize the cartridge, the Operating System initializes each of the parallel devices.

In order of device number (number 0 first), the Operating System selects each device and, if the two ID bytes are correct ($D803 contains $80 and $D80B contains $91), transfers control to the parallel device initialization routine via a JSR to the jump vector at $D819. Thus, only those devices which physically exist and have the correct ID byte values are initialized.

4.2 Parallel device initialization

After receiving control from the Operating System, the parallel device initialization routine performs device dependent initialization.

In addition, certain device independent initialization must be performed. In order to receive control to process device handler requests and low-level device I/O, the device initialization routine must set the bit in PDVMSK corresponding to the number of the device. (That bit is the lowest order bit set in SHPDVS, the device select shadow.) In order to receive control to handle device IRQ's, the device initialization routine must set the bit in PDIMSK or IPDIMK corresponding to the number of the device.

Also, if the device provides its own device handler – some parallel devices may instead rely on resident device handlers to process medium-level I/O requests – it must ensure that the name of the device is in the Operating System Device Table, HATABS, and that the corresponding handler address is that of the resident Generic Parallel Device Handler, GPDVV ($E48F).

The address of the Generic Parallel Device Handler, and not the address of the parallel device handler at $D80D, must be used because the device does not remain selected at all times. It is the responsibility of the Generic Parallel Device Handler to select the device and transfer control to the appropriate handler routine within the device ROM.

Upon completion, the parallel device initialization routine returns control to the Operating System via an RTS.

5. Device handler interface

5.1 Generic parallel device handler

The Generic Parallel Device Handler is a resident handler which is responsible for invoking the handler routines within the parallel device ROM's.

Like other device handlers, it processes medium-level I/O requests to open a device, close a device, get a byte, put a byte, return status, and perform special functions. Typically, the Generic Parallel Device Handler is called by CIO, but it may be called directly by an application, as well.

Like other resident device handlers, the Generic Parallel Device Handler is reached via an entry in HATABS which provides the address (GPDVV, $E48F) of a table of vector entries into the handler. The Operating System, itself, does not enter the Generic Parallel Device Handler into HATABS. It is not invoked unless some parallel device which supports a device handler enters it (and the device's name) into HATABS.

The Generic Parallel Device Handler routines, not knowing which of the parallel device handlers will process the request, select each of the parallel devices, in order of device number (number 0 first), and transfer control to the corresponding routine within the handler. The entry point of the parallel device handler's routine is determined from the address vector table starting at D80DH.

The C Status Flag is used to determine if the selected parallel device handler actually performed the request. Upon return to the Generic Parallel Device Handler, if the C Flag is clear, the currently selected parallel device did not perform the request and the next parallel device is selected. (Thus, even those parallel devices which do not provide a device handler must have a simple handler routine which ignores all handler requests and returns the C Flag clear.) If the C Flag is set, the request was performed and the Generic Parallel Device Handler terminates the selection process and returns to the routine which called it.

If, after calling all of the parallel device handlers, the handler request has not been performed, the Generic Parallel Device Handler returns a Non-existent Device ($82) status to the calling routine.

The entry conditions into a parallel device handler are the same as for the resident device handlers. Parameter passing is accomplished using the A, X and Y registers and the page zero IOCB. On entry, the A register contains a data byte, if necessary; the X register contains the index to the originating IOCB; and the Y register contains a Function Not Supported ($92) status.

5.2 Parallel device handler

The function of a parallel device handler is very similar to the resident device handlers.

One difference is that a parallel device handler may be called even though the request is for a different device type or a different device unit number. The parallel device handler must check the request – in the page zero IOCB or, in the case of a put-byte request, in the originating IOCB – and return the C Flag clear if it cannot handle the request.

The other difference is that the handler would not call SIO to perform the physical I/O, but instead would call a low-level I/O routine (driver) within the parallel device ROM.

After handling the request, the parallel device handler returns to the Generic Parallel Device Handler via an RTS. On exit, the A register contains a data byte, if necessary; the Y register contains the status for the request; and the C Flag should be set to indicate that the request was handled.

N.B.: Prior to beginning the device selection process, the Generic Parallel Device Handler sets the critical I/O flag (CRITIC) to disable deferred vertical blank processing. This is a bug in the Operating System, and usually has undesirable consequences. In order to compensate for this bug, any parallel device handler routine which is more than a few instructions long should store $00 into CRITIC.

6. Low-level device I/O interface

Parallel device ROM's may contain a routine similar to SIO for performing the low-level physical I/O for the parallel device.

6.1 Operating System low-level I/O

All low-level I/O requests – including those for Serial I/O – will go to the Operating System low-level I/O routine which attempts to perform the request via the low-level I/O routines within the parallel device ROM's.

Serial I/O requests are included so that the resident device handler or application need not know whether the device is a serial device or parallel device. For example, the parallel disk handler on the 1450XLD relies on the DOS to handle medium-level (CIO-type) I/O requests. The DOS makes low-level (SIO-type) I/O requests which are sometimes for a serial drive and other times for a parallel drive.

The Operating System low-level I/O routine (PIO) is reached via the jump vector at SIOV ($E459).

As in the Generic Parallel Device Handler, PIO selects each of the parallel devices, in order of device number (number 0 first), and transfers control to the low-level I/O routine within the parallel device ROM.

Control is transferred via a JSR to the jump vector at $D805. Again, the C Status Flag is used to determine if the selected parallel device low-level I/O routine actually performed the request. If a parallel device I/O routine returns the C Flag clear, then the next parallel device is selected. If a device I/O routine returns the C Flag set, the selection process is terminated and control is returned to the routine which requested the I/O. If after calling all of the parallel device low-level I/O routines the I/O request has not been performed, then the Operating System low-level I/O routine calls the resident Serial I/O routine to process the request.

To disable deferred vertical blank processing, the Operating System sets the Critical Section Flag, CRITIC, during the selection process.

The entry conditions into a parallel device low-level I/O routine are the same as for entry into the resident Serial I/O routine. All parameters passed are contained in the Device Control Block (DCB, $0300).

6.2 Parallel device low-level I/O

The parallel device low-level I/O routine (driver) performs requests for physical I/O for a parallel device. Because the request may be for a different device type or a different device unit number, the parallel device low-level I/O routine must check the request – bus ID and unit number in the DCB – and return the C Flag clear if the I/O request is for some other device.

If the parallel device ROM provides both a CIO-level handler and an SIO-level driver, it is recommended that the ROM incorporate a single routine for actually performing the physical I/O. This routine would not check the request for the correct device type and physical device unit number. Both the parallel device handler (if present) and the parallel device low-level I/O routine, which do check the request for validity, could then call this routine to perform the validated I/O request.

After performing the request, the parallel device low-level I/O routine returns to the Operating System low-level I/O routine via an RTS. On exit, the I/O has been initiated – or completed, if the device is not interrupt-driven – the Y register contains the status for the request, and the C Flag should be set to indicate that the request was processed.

N.B.: The parallel device low-level I/O routine is not allowed to modify the DCB, except for DUNIT ($0301). The original value of DUNIT is saved by PIO before the parallel device low-level I/O routine is called, and it will be restored after the parallel device low-level I/O routine returns. This restoration takes place only at the very end of PIO, so that if a parallel device low-level I/O routine changes DUNIT and then returns with Carry clear, subsequent parallel device low-level I/O routines and SIO will see the modified value of DUNIT.

7. Device IRQ handler interface

7.1 Operating System IRQ processing

When an IRQ occurs, after checking for a serial input IRQ, the Operating System checks PDVI and IPDVI to see if a parallel device initiated the IRQ.

If a parallel device initiated the IRQ and the bit corresponding to that device is set in PDIMSK or IPDIMK (whichever is appropriate), the Operating Systems transfers control to the IRQ handler of the parallel device which initiated the IRQ.

If more than one device initiated the IRQ, only the lowest-numbered device will get control (IRQ's from higher-numbered devices remain pending). Transfer of control is accomplished via a JSR to the jump vector at $D808.

Because a parallel device may be executing when the interrupt occurs, before selecting a device to handle the IRQ, the Operating System places the value of the device select shadow, SHPDVS, on the stack. When all parallel device IRQ's have been handled, the value of the device select from the stack is restored to PDVS (and SHPDVS).

After the parallel device IRQ handler returns control, the Operating System returns to the interrupted routine via an RTI.

7.2 Parallel device IRQ handling

The parallel device IRQ handling routine processes the interrupt. After the interrupt has been processed, the parallel device IRQ handler returns to the Operating System via an RTS instruction.

The Operating System support of the parallel bus is designed so that parallel I/O and serial I/O are able to be done concurrently. Therefore, in order to avoid the loss of serial port data, interrupts from parallel devices must be cleared and interrupts enabled within about 150 microseconds.

N.B.: Earlier revisions of this document stated that a parallel device IRQ routine could re-enable interrupts by executing a CLI instruction. Those earlier revisions are wrong.

A parallel device IRQ routine must not enable interrupts.

8. RAM availability

8.1 Page $D6xx and $D7xx RAM

512 bytes of RAM – addressed $D600 through $D7FF – are available for use by parallel device handlers and drivers. This RAM is not used at all by the Operating System; in particular, it is not zeroed during cold-start or warm-start.

Each card slot has a portion of this RAM allocated to it, according to the following scheme:

Memory Address	Usage
$D600 - $D61F	Slot 0 RAM
$D620 - $D63F	Reserved for use by modem devices
$D640 - $D67F	Slot 1 RAM
$D680 - $D6BF	Slot 2 RAM
$D6C0 - $D6FF	Slot 3 RAM
$D700 - $D73F	Slot 4 RAM
$D740 - $D77F	Slot 5 RAM
$D780 - $D7BF	Slot 6 RAM
$D7C0 - $D7FF	Slot 7 RAM

As the table indicates, slots 1-7 each own 64 bytes of RAM, while slot 0 owns 32 bytes of RAM. There are 32 bytes reserved for use by modem devices because 64 bytes is not enough to provide the buffering that a modem requires.

Obviously, a parallel device handler which uses more than 32 bytes of page $D6xx and $D7xx RAM cannot be placed in slot 0. With this one exception, parallel device handlers should be designed so that they will function properly in any slot. In particular, before accessing page $D6xx and $D7xx RAM, the handler must determine which slot it is in (by examining SHPDVS) and do an address calculation.

8.2 Zero-page RAM

There are 11 bytes on page zero which can be used by parallel device handlers and drivers (except during IRQ processing):

Label	[Address, Size]
STATUS	[$30, 1 byte]
CHKSUM	[$31, 1 byte]
BUFRLO	[$32, 1 byte]
BUFRHI	[$33, 1 byte]
BFENLO	[$34, 1 byte]
BFENHI	[$35, 1 byte]
BUFRFL	[$38, 1 byte]
RECVDN	[$39, 1 byte]
XMTDON	[$3A, 1 byte]
CHKSNT	[$3B, 1 byte]
NOCKSM	[$3C, 1 byte]

These 11 bytes are normally reserved for use by SIO. Since it is not possible for SIO to be active at the time that a parallel device handler or driver is called (except during IRQ processing), parallel device handlers and drivers may use these locations freely; their original values do not have to be saved and restored.

In addition, there are 4 bytes on page zero that are reserved for use by parallel device IRQ routines:

Label	[Address, Size]
ABUFPT	[$1C, 4 bytes]

The only part of the OS which uses these 4 bytes is the immediate IRQ handler. Parallel device IRQ routines need not save and restore the original values of these memory locations.

Parallel device IRQ routines which need more than 4 bytes of zero-page RAM should use the 11 SIO bytes listed above; naturally, the IRQ routine must save and restore the original values of these bytes.

Parallel device IRQ routines needing more than 15 bytes of zero-page RAM, and parallel device non-IRQ routines needing more than 11 bytes of zero-page RAM, can get more by saving and restoring other locations on page zero; the safest spot to use for this purpose is probably the zero-page IOCB ([$20, 12 bytes]).

8.3 Other RAM

The following memory locations may be used as scratch storage by non-IRQ parallel device routines. They are normally reserved for use by SIO.

Label	[Address, Size]
CDEVIC	[$023A, 1 byte]
CCOMND	[$023B, 1 byte]
CAUX1	[$023C, 1 byte]
CAUX2	[$023D, 1 byte]
TEMP	[$023E, 1 byte]
ERRFLG	[$023F, 1 byte]
CRETRY	[$029C, 1 byte]
DRETRY	[$02BD, 1 byte]
TIMFLG	[$0317, 1 byte]
STACKP	[$0318, 1 byte]
TSTAT	[$0319, 1 byte]

8.4 Vectors

Parallel device handlers should provide, in page $D6xx and $D7xx RAM, whatever vectors are necessary to replace the parallel device handler with a RAM-resident handler. Prime candidates for vectoring are the IRQ entry point (through $D808) and the low-level entry point (through $D805).

There is no uniform scheme for vectoring due to the fact that any program which intercepts one of these vectors is going to be extremely device-specific anyway.

8.5 A false-address warning

Page $D5xx is used for hardware locations within cartridges. Therefore, when parallel device handlers access their RAM on pages $D6xx and $D7xx, they must be careful not to generate any references to page $D5xx.

Due to a quirk in the 6502 microprocessor, this is not a simple as it sounds. Whenever indexed addressing is used to cross a page boundary, the 6502 will generate an extra memory cycle during which it references the memory location one page below the desired address.

For example, consider:

```
LDX #$FE
LDA $D510,X
```

During execution of the LDA instruction, the 6502 will generate a reference to memory location $D50E, as well as to the desired address $D60E.

This applies not only to LDA instructions, but to all instructions (including stores). It also applies to indirect indexed addressing, as in `LDA (ZPAGE),Y`.