# The Software Implementation of Parallel Device Handlers and Drivers, 1984-05-10 Version, Revision A

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:** | NORDIN.00332.DOC extracted from CEO.01JUN84 |
| **Title of document:** | The Software Implementation of Parallel Device Handlers and Drivers, 1984-05-10 Version, Revision A |
| **Author(s):** | Rick K. Nordin (Richard K. (Hud) Nordin) |
| **Original file date:** | 1984-05-10 |
| **Type of document:** | Memo |
| **Target audience:** | Internal |
| **Status:** | Very Advanced Draft |
| **Reference (Atari):** | (unknown) |
| **Reference (Laurent Delsarte):** | For any discussion, this PDF has been given the reference **BKUP-1984-05-10-MEMO-0010A-7** which should be quoted in any communication. |
| **Tags:** | #Atari #8bit #6502 #600XL #800XL #Parallel #PBI #Handler #Driver |

## Comments:

This version is identical to the previous one, with the exception of section “8. RAM availability”, which has been expanded.

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, 1984-05-10 Version, Revision A

R. K. Nordin
Revision A
May 10, 1984

## Table of Contents

# 1. Overview

The Atari 600XL and 800XL computers provide a parallel bus. Parallel bus devices 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 device numbers 0, 1, 6, and 7 are reserved for Atari.

The ROM 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 is accomplished by setting the corresponding bit in hardware register $D1FF. For instance, storing $04 at $D1FF selects device 2. (If more than one bit is set, the device with the lowest corresponding number is selected.) 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 reinstating 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<br>Value = $80 |
| **$D804** | Optional | Name or Type |
| **$D805 - $D807** | **Mandatory** | Low-level I/O Vector<br>Value = JMP address (low byte, high byte) |
| **$D808 - $D80A** | **Mandatory** | IRQ Handler Vector<br>Value = JMP address (low byte, high byte) |
| **$D80B** | **Mandatory** | ID Number 2<br>Value = $91 |
| **$D80C** | Optional | Device Name<br>Value = Device Name in ASCII |
| **$D80D - $D80E** | **Mandatory** | Device Handler Open Vector<br>Value = address-1 (low byte, high byte) |
| **$D80F - $D810** | **Mandatory** | Device Handler Close Vector<br>Value = address-1 (low byte, high byte) |
| **$D811 - $D812** | **Mandatory** | Device Handler Get-Byte Vector<br>Value = address-1 (low byte, high byte) |
| **$D813 - $D814** | **Mandatory** | Device Handler Put-Byte Vector<br>Value = address-1 (low byte, high byte) |
| **$D815 - $D816** | **Mandatory** | Device Handler Status Vector<br>Value = address-1 (low byte, high byte) |
| **$D817 - $D818** | **Mandatory** | Device Handler Special Vector<br>Value = address-1 (low byte, high byte) |
| **$D819 - $D81B** | **Mandatory** | Initialization Vector<br>Value = JMP address (low byte, high byte) |
| **$D81C** | Optional | Not Used<br>Value = 0 |

## 3. General device selection process

Four registers are used in the parallel device selection process:
1. PDVS/PDVI ($D1FF), Parallel device select/IRQ hardware register
2. SHPDVS ($0248), Parallel device select shadow
3. PDVMSK ($0247), Parallel device mask
4. PDIMSK ($0249), Parallel device IRQ mask

In general, the device selection process is similar for all cases. The Operating System selects each device and transfers control to the appropriate routine within the device. If during this process the Operating System discovers that the remaining devices need not be selected – for instance, if the latest device selected performed the desired function – then the selection process is terminated.

Writing a value to the parallel device selection register, PDVS, causes a device or the Floating-Point Package to be selected. The value read from PDVI indicates which of the parallel devices initiated an IRQ. Bit *n* of the value read from PDVI is set only if device *n* initiated an IRQ.

Throughout the selection process – including the restoration of the Floating-Point Package when the selection process is completed – 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.

The parallel device mask, PDVMSK, is used to control the selection process in all but the case of initialization. By convention, if bit *n* of PDVMSK is set then device number *n* exists on the parallel bus and the device will be selected to process Device Handler requests or low-level I/O requests. 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 mask, PDIMSK, is used only in the case of IRQ handling. Parallel device *n* will be selected to handle its IRQ only if bit *n* is set in PDIMSK – as well as in PDVMSK. Again, the Operating System does not set PDIMSK. It is the responsibility of each device's initialization routine to set the PDIMSK bit corresponding to the device's number if the device driver is to handle IRQ's. PDIMSK is provided for software control of device selection in the case of IRQ handling. Since IRQ handling is an extremely time-sensitive process, it may be desirable to disable the IRQ handling for a parallel device.

# 4. Device initialization interface

## 4.1 Operating System initialization

During cold-start and warm-start initialization, after initializing resident device handlers and the I/O routines, and before attempting to initialize the cartridge[1], 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, to process low-level device I/O, and to handle device IRQ's, 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 also set the bit in PDIMSK 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 device handler requests – it must ensure that the name of the device is in the Operating System Device Table, HATABS, and that the corresponding handler address in HATABS 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, 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.

1 [[No need to specify “which cartridge” because the OS used in the 600XL & 800XL & subsequent models no longer checks for the presence of a “B” or “Right” Cartridge, as on the Atari 800. There is now only one cartridge slot recognized by the OS, the one formerly known as the “A” or “Left” Cartridge.]]

# 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, and transfer control to the corresponding routine within the handler. The entry point of the parallel device handler's routine is determined from the vector table starting at $D80D.

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 (or the originating 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 that of 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 is set to indicate that the request was handled.

# 6. Low-level device I/O interface

All parallel device ROM's will 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 (PIO) 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, some parallel devices may not provide a parallel device handler, but instead rely on the resident handler for that device type. The resident handler will make low-level (SIO type) I/O requests which may sometimes be for a serial device and other times for a parallel device.

Typically, the request is made by a resident device handler, but it may come from an application, as well. 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, 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 (SIO) 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 SIO. 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 it cannot perform the I/O request.

If the parallel device ROM provides both a handler and a driver, it is recommended it 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 PIO via an RTS. On exit, the I/O has been initiated – or completed, if interrupts are not used – the Y register contains the status for the request, and the C Flag is set to indicate that the request was processed.

# 7. Device IRQ handler interface

## 7.1 Operating System IRQ processing

When an IRQ occurs, after first checking for a serial input IRQ, the Operating System checks PDVI 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 both PDIMSK and PDVMSK, 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, the Operating System transfers control to them in order of device number (number 0 first). 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), reselecting the previous device or the Floating-Point Package.

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. It is the responsibility of the IRQ handler to enable interrupts as soon as possible via a CLI instruction. 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 can be done concurrently. Therefore, in order to avoid the loss of serial data, interrupts from parallel devices must be cleared and interrupts enabled within about 150 microseconds.

# 8. RAM availability

512 bytes of address space – $D600 through $D7FF – are reserved for use by parallel device handlers and drivers. It is the responsibility of the device to supply the RAM in this area, if it does not exist on the target machine.

The first 256 bytes – $D600 through $D6FF – are reserved for use by Atari.

The second 256 bytes – $D700 through $D7FF – are divided equally among the 8 possible parallel devices, 32 bytes per device. For example, the RAM reserved for device number 1 is $D720 through $D73F. It is the responsibility of routines which need to access this RAM to calculate the address of the 32-byte block which is reserved for that parallel device. (The number of the parallel device is the lowest order bit set in SHPDVS.) All of this RAM is zeroed during initialization, prior to parallel device initialization.

The current allocation of the $D600 through $D6FF RAM which is reserved for Atari for use by planned on-board devices is as follows:

| Memory Address | Usage |
|---|---|
| **$D600 - $D61D** | Parallel Bus Disk |
| **$D620 - $D68C** | Parallel Bus Modem |
| **$D68D - $D6A4** | Parallel Bus Disk |
| **$D6C7 - $D6FF** | Parallel Bus Voice |

Note, however, that all of $D600 through $D6FF is reserved for Atari; this allocation is very subject to change and is only provided for reference.

Further RAM may be obtained through techniques used by other non-resident device handlers. The main techniques are as follows:

1. Saving in the stack and restoring carefully selected zero-page RAM.
2. Making a dynamic allocation at initialization time by altering MEMLO.
3. If the handler replaces a resident device handler, using the RAM assigned to the resident handler.
4. Negotiating with an application, through the use of appropriate commands and statuses, for RAM which is under control of the application.

See Chapter 9, “Adding New Device Handlers/Peripherals”, the section dealing with Resource Allocation, in the “ATARI Home Computer Operating System User's Manual” (C016555), for additional details.