Notes on the Altos 586 Computer

Work in progress, Jule 2022
by Lubomir Rintel

I've recently obtained the system board from the Altos 586 computer. Unfortunately, it came without documentation, the hard drive or the controller board.

With the goal of putting the system to some use I've decided to examine the hardware and firmware, learn how the system is put together and perhaps eventually substitute the missing parts.

This document contains some notes from my investigation, provided in hope of being useful to other owners of Altos 586 systems.

1. Altos 586 vs. 586T

There seems to be some documentation available for a related system, Altos 586T. While there's no service manuals or schematics, there's a reference manual[1] that describes many aspects of the computer architecture and firmware itnerfaces.

The Altos 586 system seems to be largely similar to Altos 586T. The "T" in the latter system's designation seems to stand for "Tape", as, unlike the 586, 586T includes a tape drive.

The disk controller board used in 586T seems to be vastly different from the controller board used in the Altos 586. While the Altos 586 board is bases on around the 8089 I/O controller and only controls a hard drive, the 586T board includes a Floppy and Tape controllers and utilizes a Z80 processor to handle the I/O.

Considering the above, the 586T manual is largely relevant to the 586, except for when it comes to disk I/O, be it floppy access or hard drive access.

[1] *586T/986T System Reference Manual*, P/N 690-15813-002, April 1985

2. Z80 Peripheral Controller

As on 586T, the peripheral controller is a Z80 controlling a bunch of serial interfaces, timers for baud rate generators and a RTC. It's equipped with a PIO for generating utility signal lines.

On a 586 it also has an extra duty of controlling the floppy drives, described below.

2.1. Peripherals

The peripherals are mostly the same as in the 586T machine aside from the addition of the Floppy Controller and the related circuitry.

Consult the *586T/986T System Reference Manual* for description of the peripherals common to 586 and 586T.

2.2. I/O Port Map

Address	Name	Part	Description
00h			Bus Address Top Bits Latch
20h – 23h	PIT 0	Intel 8254	Programmable Interval Timer
24h – 24h	PIT 1	Intel 8254	Programmable Interval Timer
28h – 2Bh	SIO 0	Zilog Z8440APS (Z80A SIO/0)	Serial I/O Controller
2Ch – 2Fh	SIO 1	Zilog Z8440APS (Z80A SIO/0)	Serial I/O Controller
30h – 33h	SIO 2	Zilog Z8440APS (Z80A SIO/0)	Serial I/O Controller
34h – 37h	PIO	Zilog Z8420APS (Z80A PIO)	Parallel I/O Controller
38h – 3Bh	FDC	Western Digital FDC1797	Floppy Disk Controller/Formatter
3Ch	DMA	Zilog Z8410APS (Z80A DMA)	Direct Memory Access Controller
40h			DMA Carrier/Parity
80h – 9Fh	RTC	MM58167AN	Microprocessor Real Time Clock

2.3. Floppy Disk Controller/Formatter

The floppy disk controller is the FDC1797 part hooked on in a straightforward manner. The output control signals are interfaced to a standard Shuggart floppy connector via open-collector drivers, the input signals go through a line drivers with external pull up register array. The read data signal is passed through external data separator circuitry to generate clock and data from MFM encoded signal.

Some of the FDC lines involved in the Floppy Drive access are connected to the PIO. In particular, the *#DDEN* pin needs to be set low in order to read double-density (MFM) disks, which is almost certainly the case. The*INTRQ* pin is connected to PIO' s *PA7* input pin. No idea if it's actually used.

To actually access the floppy drive, the appropriate *Drive Select* signal needs to be pulled low. For reading double-sided floppies, the drive head/side is selected by the *Side Select* signal. The *Drive Select* and *Side Select* signals are also both controlled by the PIO.

For the details about the control lines consult the paragraph on the Parallel I/O Controller below.

2.4. Parallel I/O Controller

For the most part, the pins I've traced out are related to the floppy circuitry. Others are unknown.

#	Pin	Direction	Description
15	PA0		
14	PA1		
13	PA2		
12	PA3		
10	PA4		
9	PA5		
8	PA6		
7	PA7	Input	FDC INTRQ (Pin 39)
27	PB0	Output	"586 sync" sub-board (Pin 7)
28	PB1	Output	"586 sync" sub-board (Pin 8)
29	PB2	Input	Goes into unpopulated connector next to FDD (Pin 10)
30	PB3	Output	Floppy Drive Select 0
31	PB4	Output	Floppy Drive Select 1
32	PB5	Output	Floppy Side Select
33	PB6	Output	FDC #DDEN (Pin 37)
34	PB7	Input	PIT ? OUT2 (Pin 17)

3. Floppy Interface

3.1. Geometry and Format

The Altos 586 computer uses 5¼" diskettes in a drive with following parameters:

Data density	Double Density
Encoding	MFM
Speed	300 RPM
Bit rate	250 kbps
Tracks	80
Heads/Sides	2

The floppies XENIX, CP/M and the firmware use the following format (though the controller permits other formats).

Sector size	512 bytes
Sectors per track	9

Note that while this format is fairly unusual for 5¼" drives and media, it's exactly the same as IBM PC 3½" DD (720 KB) disks use. Regular 3½" drives can be used if they're adapted for the Altos 586 floppy connector pinout (see below).

The regular 1.44 MB floppies can usually be used with DD format, but the density select window needs to be covered with a piece of opaque tape.

3.2. Floppy Connector Pinout

The floppy connector is not PC compatible and therefore usual PC floppy drives can't be used without modifications. It uses straight ribbon cables, not the PC-style ones with a twist.

The pinout (and drive geometry) is fairly close to to what Commodore Amiga computers use. There seems to be a plenty of tutorials on modifying PC drives for Amiga on the Web. The same modificaqtions should work for the Altos 586.

Function	Pin		Direction	Function	Note
	1	2	Output	Motor Enable	
	3	4			
	5	6			
	7	8	Input	Sector Index	
	9	10	Output	Disk Select 0	
	11	12	Output	Disk Select 1	
	13	14			
	15	16	Output	Motor Enable	Jumpered with E32
GND	17	18	Output	Step Direction	
	19	20	Output	Step Pulse	
	21	22	Output	Write Data	
	23	24	Output	Write Gate	
	25	26	Input	Track 0	
	27	28	Input	Write Protect	
	29	30	Input	Read Data	
	31	32	Output	Side/Head Select	Jumpered by E1, closed for double-sided drive
	33	34	Input	Drive Ready	

4. Firmware Interface

The firmware interface is unsurprisingly very similar to 586T. A notable difference is addition of the Floppy Channel, described below.

The *586T/986T System Reference Manual* is not too clear about the offset of various register blocks though they can be for the most part inferred from their respective lengths, since they immediately follow each other.

The layout of register blocks on the 586 is somewhat different due to the inclusion of the *Floppy Drive Interface Registers* block before the *Time of Day Registers*. offsets of each block. It is incidentally also the offset of each command register the firmware looks at, since each block starts with a command register.

Offset	Description
0Ah	Communication Channel Registers (Port 1)
20h	Communication Channel Registers (Port 2)
36h	Communication Channel Registers (Port 3)
4Ch	Communication Channel Registers (Port 4)
62h	Communication Channel Registers (Port 5)
78h	Communication Channel Registers (Port 6)
8Ah	Floppy Drive Interface Registers
D4h	Time of Day Registers

The *Communication Channel Registers* and *Time of Day Registers* are described in the *586T/986T System Reference Manual*, respectively. The rest of this section deals with the Floppy interface exclusively.

4.1. Floppy Drive Interface Registers

This structure is present at offset 8Ah from the command control block (CCB). Not all members are used by all commands.

Offset	Size	Description	Typical Value
+00h	BYTE	Command	87h, 88h, . . .
+01h	BYTE	Status	00h, 48h, 40h, C0h, . . .
+02h – +04h	BYTE[3]	Next Floppy Command Pointer	Pointer to *Floppy Command Queue* entry
+05h	BYTE	Floppy Command Queue Size	02h
+06h	BYTE	Last Floppy Command Index	01h
+07h	BYTE	Next Floppy Command Index	00h
+08h	BYTE	Unused	00h
+09h	BYTE	Unused	00h
+0Ah – +B3h	BYTE[32]	Floppy 1 Parameters	*Floppy Parameters*
+B4h – +D3h	BYTE[32]	Floppy 2 Parameters	*Floppy Parameters*

4.2. Floppy Drive Interface Commands

There factory software has been seen issuing three different command, with two of them having known function:

Command	Description
87h	Set Floppy Drive Parameters
88h	Submit Floppy Command Queue
8fh	Unknown

As with the other command blocks, the most significant bit indicates to the peripheral controller that there's a new command to process. Once the processing is finished, the peripheral controller sets the bit to zero.

4.2.1. Set Floppy Drive Parameters

The *Set Floppy Drive Parameters* command reads the *Floppy Parameters* structure (see below) and saves it in the controller's SRAM. The other registers in the block are ignored. This command needs to be called to set the sector size before any Floppy I/O.

Failure to set the sector size up will cause the read commands to overflow a data buffer, corrupting the controller SRAM. The main processor firmware calls this command on startup.

4.2.2. Submit Floppy Command Queue

The *Submit Floppy Command Queue* command submits the *Floppy Command Queue* for processing. Said queue is in fact a ring buffer whose entries are the addresses of *Floppy Disk Interface Parameter Block* structures (described below).

Upon submission, the processing starts with the entry on address in *Next Floppy Command Pointer*. an entry, it increments the next entry index, wrapping it around the queue size if necessary. The processing finishes once the next entry index is equal to the last entry index.

Note that the I/O processor modifies the *Next Floppy Command Index*register when it finishes handlin an entry, but keeps the value in *Next Floppy Command Pointer* unchanged, while still using it to calculate the address of the next entry to process. When you submit another batch of commands When you submit another batch of commands you need to adjust it accordingly.

This command doesn't use the *Floppy Parameters* in the register block, though it requires that the parameters are set by the *Set Floppy Drive Parameters* previously.

4.3. Floppy Parameters

Used to issue I/O commands, not used by command.

Offset	Size	Description	Typical Value
+00h	WORD	Unknown	0200h
+02h	WORD	Sector size	0200h
+04h	BYTE	Unknown	06h
+05h – +1fh	BYTE[. . .]	Unknown/Filler (00h)	00h . . .

4.4. Floppy Command Queue

Offset	Size	Description	Typical Value
+00h	DWORD	Floppy Command Block Address	Pointer to *Floppy Command Block*
+04h	DWORD	Floppy Command Block Address	Pointer to *Floppy Command Block*
. . .			
+04h * n	DWORD	Floppy Command Block Address	Pointer to *Floppy Command Block*

4.5. Floppy Disk Interface Parameter Block

Offset	Size	Description	Typical Value
+00h	High 4 bits	Command	10h, 20h, 1Ah, 2Ah, . . .
	Low 4 bits	Retries	
+01h	BYTE	Status	
+02h	BYTE	Drive Number	00h
+03h	BYTE	Track	00h – 28h
+04h	BYTE	Head	00h, 01h
+05h	BYTE	Sector	01h – 09h
+06h – +09h	BYTE[3]	Data Buffer Address	

5. Reading a Floppy from the Firmware ROM Monitor

This section provides a concrete example of using the Floppy Interface of the peripheral controller.

```
FAILED POWER-UP TEST B   Controller board error. I got no board.
Monitor Version a2.2
Enter [1] to boot from Hard Disk
Enter [2] to boot from Floppy Disk
Enter [3] to enter Monitor
Enter option: 3
```

First let's find the Channel Control Block address (CCB). Read it from 1FFFCh:

```
< A, B, D, G, I, K, L, M, O, R, S, X > D 1000:FFFC 4
  1000:FFFC                              16040000   *................*
```

It's 416h. The channel register offsets are calculated from this address. The new command register is at offset 05h from the beginning of the CCB, therefore at address 41Bh.

Now let's prepare a small program that just bumps the command counter so that the controller firmware will know it needs to check for new commands. It then invokes to return to the monitor. We'll place it at address 2000h, which is an arbitrarily chosen address in the DRAM.

It's not strictly necessary here, since each key press results in a serial command and the command registers are checked anyway, but would be necessary for floppy communication outside the monitor.

```
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:2000
  0000:2000  00-fe              incb 0x41b
  0000:2001  00-06
  0000:2002  00-1b
  0000:2003  00-04
  0000:2004  00-cc              int 3
  0000:2005  00-,
```

We can proceed setting the floppy parameters. This is also not strictly necessary if we're using the ROM monitor, because the firmware must have initialized the parameters for us.

Let's do it anyway though, for demonstration purposes. It might also be useful if the floppy format uses a sector of size other than 512 bytes.

The floppy command register is at offset 8Ah from the beginning of the CCB, therefore at address 4A0h.

For reasons explained below, we start setting the bytes from 4A1h and set the command byte (4A0h) last.

```
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:04A1
  0000:04A1  00-00              Status bit, doesn't matter
  0000:04A2  60-60              Buffer pointer, not used
  0000:04A3  09-09
  0000:04A4  00-00
  0000:04A5  02-02
  0000:04A6  00-00              Pointer counter, not used
  0000:04A7  00-00
  0000:04A8  00-00              Unknown
```

```
  0000:04A9  00-00
  0000:04AA  00-00              Floppy 1 parameters
  0000:04AB  02-02
  0000:04AC  00-00
  0000:04AD  02-02
  0000:04AE  06-06
  0000:04AF  00-00
      ...
  0000:04C9  00-00
      ...
  0000:04CA  00-00              Floppy 2 parameters
  0000:04CB  02-02
  0000:04CC  00-00
  0000:04CD  02-02
  0000:04CE  06-06
  0000:04CF  00-00
  0000:04E9  00-00
  0000:04EA  11-,
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:04A0
  0000:04A0  00-87              The "Floppy Parameters" command byte
  0000:04A1  00-,
```

We set the command byte last to make sure the controller firmware doesn't see a valid command until the parameters are all set.

Now we can bump the command counter with the program we entered in at the beginning (though, as stated above, this is not really necessary):

```
< A, B, D, G, I, K, L, M, O, R, S, X > G 0:2000
Break ....
CS:IP FC00:0000  Flags  - - - - S - - - -
  AX    BX    CX    DX    SI    DI    DS    ES    SS    SP    BP
  0000  0000  0000  0000  0000  0000  0000  0000  0000  0FFC  0000
```

Let's enter some actual floppy I/O requests now.

Start with a seek:

```
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:2100
  0000:2100  00-10              Seek command
  0000:2101  00-00              Status
  0000:2102  00-00              Unknown, 00h
  0000:2103  00-05              Track
  0000:2104  00-01              Head
  0000:2105  00-05              Sector
  0000:2106  00-,
```

Seek to a different track, so that we're sure we see the head move:

```
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:2200
  0000:2200  00-10              Seek command
  0000:2201  00-00              Status
  0000:2202  00-00              Unknown, 00h
```

```
  0000:2203  00-25              Track
  0000:2204  00-01              Head
  0000:2205  00-05              Sector
  0000:2206  00-,
```

And then read a sector. The controller will seek back to the appropriate track:

```
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:2300
  0000:2300  00-10              Read command
  0000:2301  00-00              Status
  0000:2302  00-00              Unknown, 00h
  0000:2303  00-05              Track
  0000:2304  00-01              Head
  0000:2305  00-05              Sector
  0000:2306  00-00              Destination buffer address: 00002500h
  0000:2307  00-25
  0000:2308  00-00
  0000:2309  00-00
  0000:230A  00-,
```

Create a queue with the three commands we created above:

```
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:2400
  0000:2400  00-00              First command at 00002100h
  0000:2401  00-21              (See above: seek to track 5)
  0000:2402  00-00
  0000:2403  00-00
  0000:2404  00-00              First command at 00002200h
  0000:2405  00-22              (See above: seek to track 25h)
  0000:2406  00-00
  0000:2407  00-00
  0000:2408  00-00              First command at 00002300h
  0000:2409  00-23              (See above: read from CHS 5/1/5 to 2500h)
  0000:240A  00-00
  0000:240B  00-00
  0000:240C  00-,
```

Now invoke the queue the same way as we set the parameters:

```
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:04A1
  0000:04A1  00-ff              Status
  0000:04A2  60-00              The queue is at 00002400h
  0000:04A3  09-24
  0000:04A4  00-00
  0000:04A5  02-04              Queue Length
  0000:04A6  00-03              Last valid command is 03h commands in the queue
  0000:04A7  00-00              Start with the command 00h
  0000:04A8  00-00
  0000:04A9  00-00
  0000:04AA  00-,
< A, B, D, G, I, K, L, M, O, R, S, X > A 0:04A0
  0000:04A0  00-c8              Now set the command byte
  0000:04A1  48-,               to process the queue
```

```
< A, B, D, G, I, K, L, M, O, R, S, X > G 0:2000
Break ....
CS:IP FC00:0008  Flags  - - - - S - A - -
  AX    BX    CX    DX    SI    DI    DS    ES    SS    SP    BP
  0000  0000  0000  0000  0000  0000  0000  0000  0000  0FF8  0000
```

The command is in. Let’s keep checking the status byte until it indicates success:

```
< A, B, D, G, I, K, L, M, O, R, S, X > D 0:04A1 1
  0000:04A1    48                                  *.H.$...........*
```

The status is now 48h. We need it to be 40h. Try some more:

```
< A, B, D, G, I, K, L, M, O, R, S, X > D 0:04A1 1
  0000:04A1    40                                  *.@.$...........*
```

Now the status is 40h, which indicates no error.

The sector data is now at 2500h, let’s check it out:

```
< A, B, D, G, I, K, L, M, O, R, S, X > D 0:2500 200
  0000:2500  EB080000 30000000 0002CCEB FD000000   *k...0.....Lk}...*
      ...
  0000:26F0  00000000 00000000 00000000 00000000   *................*
< A, B, D, G, I, K, L, M, O, R, S, X >
```

5.1. Z80 ROM Monitor

The factory firmware seems to include a ROM Monitor software. Having a firmware monitor is very helpful for investigation of the system. It would allow controlling the Z80 peripherals directly and operating them without the main 8086 processor.

Unfortunately, I didn't discover a way to enter the factory ROM monitor. I've ended up temporarily replacing the ROM with a different one, allowing me to poke around the controller. That way I've lost the way of utilizing the factory firmware on the main processor, which would just end up being stuck in a tight loop.

6. Hard Drive Controller Board

I don't own this board and know little more about the board than could be observed from firmware disassembly.

This hard drive controller is based on the Intel 8089 I/O processor. The 8089 processor is connected to onboard logic and sector buffer SRAM via a private bus, allowing them to be accessed without congesting the system bus. It is also connected to the main memory via the system bus, allowing for efficient DMA transfers of the sector data.

The 8089 is controlled with specialized programs placed in the main memory and synchronized with the main CPU with interrupts.

The firmware includes one such program to handle the disk reads and writes. It is parametrized by I/O parameter blocks (IOPB) placed in the main memory. It is described in detail in section below.

6.1. IOPB Handling Program

This program is provided in the firmware image and run on the 8089 on the hard drive board to service the I/O operations.

It is controlled by the I/O parameter block structure to ultimately transfer data between a main memory buffer and the hard drive. The data is not transferred directly between the controller's data port and the main memory. Instead, the data is bounced through a a buffer in controller board's SRAM.

This utilizes the main memory bus more efficiently. The controller data port transfers eight bits of data at a time at a rate determined by the disk rotation speed. These slow accesses are done over a bus that's private to the controller board, avoiding congestion of the main bus. On the other hand, the SRAMs allows 16-bit accesses at a much faster rate, utilizing the main memory as little as possible.

On reads, the data is first transferred from the data port to the SRAM buffer and then from SRAM to the main memory. On writes, the data is copied from main memory to SRAM buffer and then from SRAM to the data port.

The number of sectors specified in IOPB determines how many transfers are done between the data port and the SRAM. The number of bytes transferred between SRAM and main memory is specified in IOPB directly. These two numbers are related -- main memory transfer needs to transfer sufficient number of bytes to include complete sector data.

The commented I/O program follows:

```
; Program parameters, passed by the user
IOPB_OP         EQU 04H
IOPB_STATUS     EQU 05H
IOPB_CYL        EQU 06H
IOPB_DRV_HD     EQU 08H
IOPB_SEC        EQU 09H ; Starting sector
IOPB_BYTE_CNT   EQU 0AH ; The DMA buffer size
IOPB_DMA_BUF    EQU 0CH ; The DMA buffer address
IOPB_SEC_CNT    EQU 10H ; Number of sectors

; These are reserved to be used by the IOP
```

```
IOPB_IO_SIZE    EQU 12H  ; Written by the IOP
IOPB_LAST_CYL   EQU 14H  ; Last cylinder we've seeked to
IOPB_LINK_REG   EQU 16H  ; Return address of subroutine calls

; Bits of IOPB_OP
OP_BIT_RD       EQU 0    ; Read (0) or write (1)
OP_BIT_UNK      EQU 2
OP_BIT_ECC      EQU 3    ; Include the ECC bits on read
OP_BIT_SEL      EQU 5    ; Drive/Head has changed
OP_CMD_BITS     EQU 0FH

; Bits of IOPB_STATUS
STATUS_BIT_ERR  EQU 7

; These are addresses in 8089's I/O space
HDD_SRAM_ADDR   EQU 00000H
HDD_REGS_ADDR   EQU 0FFD0H

; Registers at HDD_REGS_ADDR
PORT_DATA       EQU 00H
PORT_DRV_HD     EQU 02H
PORT_04H        EQU 04H
PORT_CMD_STAT   EQU 06H

ENTRY:
        movi  gc,HDD_REGS_ADDR

        ; Is bit 5 set?
        jnbt  [pp].IOPB_OP,OP_BIT_SEL,NO_SEL

        ; If bit 5 is set, then drive and head might have
        ; CHanged and we need ensure correct head is selected.
        movbi [gc].PORT_CMD_STAT,80H
        movb  [gc].PORT_DRV_HD,[pp].IOPB_DRV_HD
B0:     jnbt  [gc].PORT_CMD_STAT,7,B0

        movbi [gc].PORT_CMD_STAT,20H
        movi  [pp].IOPB_LAST_CYL,0
C0:     jbt   [gc].PORT_CMD_STAT,0,C0

        ljnbt [gc].PORT_04H,0,RETURN_81H        ; Error?

NO_SEL:
        andbi [pp].IOPB_OP,OP_CMD_BITS  ; Command bits mask
        ljzb  [pp].IOPB_OP,RETURN_00H   ; Command is zero

        ; Select head
        movb  [gc].PORT_DRV_HD,[pp].IOPB_DRV_HD
B1:     jnbt  [gc].PORT_CMD_STAT,7,B1

        ; Seek to cylinder
        movb  [gc].PORT_DATA,[pp].IOPB_LAST_CYL
        movb  [gc].PORT_DATA,[pp].IOPB_LAST_CYL+1
```

```
        movb  [gc].PORT_04H,[pp].IOPB_CYL
        movb  [gc].PORT_04H,[pp].IOPB_CYL+1
        movbi [gc].PORT_CMD_STAT,10H
        mov   [pp].IOPB_LAST_CYL,[pp].IOPB_CYL
C1:     jbt   [gc].PORT_CMD_STAT,0,C1
D0:     jnbt  [gc].PORT_04H,1,D0

        ; Zero sectors? Tothing to do.
        jzb   [pp].IOPB_SEC_CNT,RETURN_00H

        ; A read?
        jbt   [pp].IOPB_OP,OP_BIT_RD,HDD_SRAM_XFER

        ; Whis is a write. Transfer the data from main memory
        ; DMA buffer to controller board SRAM first
        lpd   ga,[pp].IOPB_DMA_BUF
        movi  gb,PORT_DATA
        mov   bc,[pp].IOPB_BYTE_CNT
        call  [pp].IOPB_LINK_REG,MEM_SRAM_XFER

HDD_SRAM_XFER:
        ; Setup for for a transfer between SRAM and the HDD's
        ; data register, common to read and write
        movi  gb,HDD_SRAM_ADDR
        movi  mc,0FE80H; Why?
        movi  ga,HDD_REGS_ADDR+PORT_DATA
        movi  [pp].IOPB_IO_SIZE,512

        ; A read or write?
        jnbt  [pp].IOPB_OP,OP_BIT_RD,XFER_WR

        ; A read. Setup for transfer from HDD from SRAM.
        movi  cc,8A28H
        wid   8,16

        ; Go do the transfer if we don't want ECC
        jnbt  [pp].IOPB_OP,OP_BIT_ECC,XFER_SEC

        ; We want ECC. Bump the transfer size.
        movi  [pp].IOPB_IO_SIZE,512+5
        jmp   XFER_SEC

XFER_WR:
        ; A write. Setup for transfer from SRAM to HDD.
        movi  cc,5628H
        wid   16,8
        jnbt  [pp].IOPB_OP,OP_BIT_UNK,XFER_SEC
        movi  [pp].IOPB_IO_SIZE,4  ; No idea what this is.

XFER_SEC:
        ; Transfer one sector
        mov   bc,[pp].IOPB_IO_SIZE
        movb  [gc].PORT_DATA,[pp].IOPB_SEC
```

```
        xfer

        ; Check for errors
        movb  [gc].PORT_CMD_STAT,[pp].IOPB_OP
        jmcne [gc].PORT_CMD_STAT,RETURN_ERR

        ; Are we done or there's more?
        decb  [pp].IOPB_SEC_CNT
        jzb   [pp].IOPB_SEC_CNT,HDD_SRAM_XFER_DONE

        ; There's more.
        incb  [pp].IOPB_SEC
        jmp   XFER_SEC


HDD_SRAM_XFER_DONE:
        ; Transfer between HDD and SRAM has finished.
        ; Was it a write? Then we're all done.
        jnbt  [pp].IOPB_OP,OP_BIT_RD,RETURN_00H

        ; This was a read. We need to transfer from SRAM
        ; to the main memory DMA buffer.
        lpd   gb,[pp].IOPB_DMA_BUF
        movi  ga,PORT_DATA
        mov   bc,[pp].IOPB_BYTE_CNT
        call  [pp].IOPB_LINK_REG,MEM_SRAM_XFER

RETURN_00H:
        ; Successful return.
        movbi [pp].IOPB_STATUS,0
        jmp   DONE

MEM_SRAM_XFER:
        ; A subroutine that does a memory-to memory transfer.
        ; Used to copy data between the controller SRAM and
        ; the DMA buffer in main memory, in either direction.
        wid   16,16
        movi  cc,0C208H
        xfer
        nop
        movp  tp,[pp].IOPB_LINK_REG

RETURN_ERR:
        ; Error return. Communicate the error from the status
        ; register to the IOPB.
        movb  [pp].IOPB_STATUS,[gc].PORT_CMD_STAT
        andbi [pp].IOPB_STATUS,7EH
        setb  [pp].IOPB_STATUS,STATUS_BIT_ERR
        movbi [gc].PORT_CMD_STAT,00H
        jmp   DONE

RETURN_81H:
        ; Error 1.
```

```
        movi  [pp].IOPB_STATUS,STATUS_BIT_ERR+1

DONE:
        sintr
        hlt
```