Errata: FYSOS: Media Storage Devices
1st Edition
Dated: 31 Oct 2023

Page 3-9: Table 3-7:

The 0x9E command should be 0x8E, and ignore the line with command 0x1E.

4Eh	No	Restore
8Eh	No	Drive Specification
8Fh	Yes	Relative seek inward/outward
ADh	Yes	Format and Write

Page 5-1: First paragraph:

It is older drives that only supported ATA while newer drives support the ATAPI interface. The controller doesn't care either way since it only passes data on to the drive.

Table 5-7: Bit 6 is zero when a write is being performed.

Page 5-9: Second to last paragraph and code block should read:

The ATA specification states that you should read the ATA_ERROR register just after the reset, and states that the device shall clear bit 7 in the ATA_ERROR register and clear bits 6, 5, 4, 3, 2, and 0 in the ATA_STATUS register.

```
  if ((inp(base + ATA_ERROR) & 0x80) != 0)
    return FALSE;
```

Page 5-11: Added the two notes below:

CFA mode is used for Compact Flash cards and may use 8-or 16-bit reads and writes. Transfers of 8-bits were used on older hardware and may require a newer model to issue a Set Feature command to revert back to 8-bit transfers. More modern CFA cards may already be set to 16-bit transfers after reset.

The original CMD-640 and RZ-1000 IDE chips had a bug that would cause data corruption when aggressive acceleration was used.

Page 8-1: Paragraph 4:

Should be Request Sense command, not Sense Mode command.

Page 8-4: Added the following note box:

Now wait for the drive to not by busy and the DRQ bit to become set and read or write the expected count of bytes.

If your code uses interrupts instead of polling, you will need to wait for an interrupt after sending the packet and before transferring the data.

That's it. With this type of command, ...

Page 9-7: Offset 03h and 0Ah should be 1 byte in size, not 2:

Table 9-1: The DMA Bus Master Registers

The DMA Bus Master Registers				
Name	Off	Size	Type	Description
Channel 0				
BM0_COMMAND	00h	1	R/W	Command Register (Primary Bus)
BM0_RESV0	01h	1	n/a	Device Specific (Reserved)
BM0_STATUS	02h	**1**	R/WC	Status Register (Primary Bus)
BM0_RESV1	03h	1	n/a	Device Specific (Reserved)
BM0_ADDRESS	04h	4	R/W	Address Register (Primary Bus)
Channel 1				
BM1_COMMAND	08h	1	R/W	Command Register (Secondary Bus)
BM1_RESV0	09h	1	n/a	Device Specific (Reserved)
BM1_STATUS	0Ah	**1**	R/WC	Status Register (Secondary Bus)
BM1_RESV1	0Bh	1	n/a	Device Specific (Reserved)
BM1_ADDRESS	0Ch	4	R/W	Address Register (Secondary Bus)

Page 9-14: Added the following paragraph:

Waiting for DMA Transfers

For PIO transfers, after sending the command, you can use the `ata_wait()` function as described in Listing 9-2. However, for Bus Master DMA transfers, the specification states a little different technique is used.

When waiting for PIO transfers, you wait for the BSY bit to become clear and the DRQ bit to become set before you read or write to the disk's data register. For DMA transfers, the controller will clear the BSY bit and set the DRQ bit, or it will set the BSY bit and clear the DRQ bit. You have to watch for both indicators then wait for and interrupt to fire for the indication of a completed transfer.

I think the idea of setting the BSY bit and clearing the DRQ bit is backwards to what should happen, though you have to watch for it anyway since it is in the specification.

Page 9-14: Added the following note:

One more thing to know about interrupts. On a read command, command 0x20 for example, when interrupts and PIO are used, the controller will fire an interrupt after the command is sent and before the reading of the sector. This is to show that it is ready for you to start reading the sector.

Please change the read_pci() and write_pci() functions in pci.h to the following:

```
// read from the pci config space
bit32u read_pci(const bit8u bus, const bit8u dev, const bit8u func,
                                    const bit8u port, const bit8u len) {
  bit32u ret;

  const bit32u val = 0x80000000 |
    (bus << 16) |
    (dev << 11) |
    (func << 8) |
    (port & 0xFC);
  outportl(PCI_ADDR, val);
  ret = (inportl(PCI_DATA) >>
                    ((port & 3) * 8)) & (0xFFFFFFFF >> ((4-len) * 8));
  return ret;
}

// write to the pci config space
void write_pci(const bit8u bus, const bit8u dev, const bit8u func,
                    const bit8u port, const bit8u len, bit32u value) {
  bit32u val = 0x80000000 |
    (bus << 16) |
    (dev << 11) |
    (func << 8) |
    (port & 0xFC);
  outportl(PCI_ADDR, val);

  // get current value
  val = inportl(PCI_DATA);

  // make sure value is of 'len' size
  value &= (0xFFFFFFFF >> ((4-len) * 8));

  // mask out new section
  if (len < 4) {
    val &= (0xFFFFFFFF << (len * 8));
```

```
    val |= value;
  } else
    val = value;

  outportl(PCI_DATA, val);
}
```

In the HDC_TYPE.CPP file, under the ata_device_reset() function, please add the following line, marked with a '+' below:

```
  if ((error & 0x80) || (status & 0x7D))
    return FALSE;

  // force a select next time since we reset the controller above
  cur_selected = 0xFF;
+ ata_select_drv(cntrlr->base, drv, 0, 0);

  bit8u count = inportb(cntrlr->base + ATA_SECTOR_COUNT);
  bit8u number = inportb(cntrlr->base + ATA_SECTOR_NUMBER);
  if ((count == 1) && (number == 1)) {
```

Also, I found a few errors in the atapi_tx_packet_rx_data() function. Please email me and I will send you the new function code.

Page 8-4: Add the following note to the last of the page:

Please note that the amount you place in the ATA_LBA_MID_BYTE and ATA_LBA_HIGH_BYTE registers is the amount of bytes to transfer per DRQ assertion. i.e.: The count of bytes to transfer per loop you check for data ready, usually the size of a sector.

Page 10-8: Add the following note to the last of the page:

Even though a device may show that it supports the default mode of Multi-word, mode 2 DMA, as shown in the source code accompanying this book, I have found that some devices will not revert back to this mode when Ultra-DMA is supported. Therefore, if UDMA is supported, try setting to the highest mode allowed. However, please note that this does not guarantee that it will work at that mode if your PCI(e) doesn't support that mode.

hdc_type.cpp:
I have been told that some releases of the hdc_type.cpp file on the ISO may have the following line of code at or around line 101.

```
  return 0;
```

This is by mistake. It was a test run that should have been fixed before release. If this line is present, please comment or remove.