
Operating Systems Development Series

# Operating Systems Development - Enabling A20

**by Mike, 2008**

This series is intended to demonstrate and teach operating system development from the ground up.

## Introduction

Welcome! :)

In the previus tutorial, we looked at how to switch the processor into a 32 bit mode. We also learned how we can access up to 4 GB of memory. This is great--but, **how**?

Also, remember that the PC boots into real mode, which has the limitation of 16 bit registers. And, hence, 16 bit segment addressing. This limits the amount of memory you can access in real mode. Because of this, we still cannot access even up to 1 GB of memory yet. Heck, we cannot even go passed the 1 MB barrier yet! What to do? We have to enable the 20th address line. This will require direct hardware programming, so we will talk about that as well.

So, This is what's on the menu:

- Direct Hardware Programming - Theory
- Direct Hardware Programming and Controllers
- Keyboard Controller Programming - Basics
- Enabling A20
- Pizza :)

**Do to the use of high level languages, like C, being able to access more then 1 MB of memory can be critical. Because of this, enabling A20 (Address line 20) will be important!**

**Note: Remember that we cannot access over 1 MB yet! Doing so will cause a triple fault.**

Also, because we are going to go over direct hardware programming, this tutorial will be a little more complicated then previous ones. Don't worry - You will get more expeirence with direct hardware programming later when we develop device drivers for the Kernel.

*Ready?*

## Get Ready

For those who have been with me this far, I am certain you know how hard OS development is. However, we have not touched anything close to hard. All of the concepts listed here is still very basic, and yet quite advanced. However: **Things are only going to get much more harder.**

Every single controller must be programmed a special way in order to work correctly. For example, to write (or read) a hard drive, you must first determin if it is an IDE or SCSI drive. Then, you have to determin the drive number it is, and program it using either the **IDE Controller** or **SCSI Controller**, which control the IDE and SCSI connections, respecitvley. Both of these controllers are different.

To add more complexity, a "Sector" might not be 512 bytes. Hence, "Reading and writing sectors" is vauge.

Then comes memory management and fragmentation. This is where **Paging**, **Virtual Address Space**, and the **Memory Management Unit (MMU)** comes into play.

Reading and writing any drive is very different then any other drive. This is even true at the bootsector level. The typical format and file system is different between media, so code that boots from a FAT12 floppy will **Not** work to boot a CDFS Filesystem CD ROM. By Abstracting hardware specific (And low level code), we can make most of the code, however, work for these devices.

When we say "Write a file to hard drive", we normally don't want to define what a "file" is, because we shouldn't. We shouldn't have to worry about what controller to read to/from, nor the exact location on disk. **This is why abstraction is *very* important!**

Everything here is primairly for protected mode (i.e., it is 32 bit code), although it will work in real mode as well. Because of this, remember the rule of protected mode:

- **No interrupts are avilable! The use of any interrupt will cause a triple fault**

...Hence, you are **Completely** on your own.

# Kernel Debugging

Debugging is an art form. It provides a way of trapping problems, and fixing errors through software before they become serious. **Kernel Debugging** relates to debugging kernel-level Ring 0 programs. This is never an easy task.

## Debuggers in High Level Languages

Most debuggers in languages, like C and C++, provide a way of displaying varable and routine names, and their values, locations, etc during runtime. The problem? We don't have any symbolic names yet in any of our programs. We are still working at the **Binary** level.

What this means is that we need a debugger that could work and display memory directly. Bochs has a debugger just for us.

## Bochs Debugger

Bochs comes with a debugger called **bochsdbg.exe**. When you launch it, you will be given the same startup screen from Bochs.exe. Load your configuation file, and start the emulation.

Bochs debugger and display window will now appear, and you should see the line:

```
[0x000ffff0] f000:fff0 (unk. ctxt): jmp f000:e05b        ; ea5be000f0
< bochs:1> _
```

In the second line, bochs tells you the number of commands sent to it (In this case, this is the first command, so a 1 is displayed). You can type your commans here.

The first line is the important line. It tells you the current instruction, absolute address, and seg:offset address. It also gives you the machine language **Operation Code (Opcode)** equivelant.

## HELP command

The **help** command gives you a list of available commands.

## BREAK command

The **b (break)** command allows you to set breakpoints at addresses in memory. For example, if we are trying to debug our OS, we need to start at the bootloader (07c00:0). However, Bochs Debugger starts where the BIOS is at. Because of this, we will need to set a breakpoint to 0x7c00:0, and continue execution until that breakpoint is reached:

```
                    // BIOS is at 0xea5be000f0
[0x000ffff0] f000:fff0 (unk. ctxt): jmp f000:e05b
< bochs:1> b 0x7c00                  // Sets the breakpoint to 0x7c00:0
< bochs:2> c                         // Continue executon
< 0> Breakpoint 1, 0x7c00 in ?? < >     // Our breakpoint is hit
Next at t=834339
                    // We are now at our bootloaders first instruction
< 0> [0x00007c00] 0000:7c00 (unk. ctxt): jmp 7cb5        ; e9b200/DIV>
```

The above tells us that our main() function in our bootloader is at 0x7cb5. This makes sinse because, remember that the OEM Parameter Block is between this jump instruction, and the start of main().

Knowing that the bootloader loads stage 2 at 0x500, lets break to it:

```
< bochs:3> b 0x500
< bochs:4> c
< 0> Breakpoint 2, 0x500 in ?? < >
Next at t=934606
<0> [0x000000500] 0050:0000 (ink. ctxt): jmp 00a0       ; e99d00
< bochs:5> _
```

Now, we are at the beginning of stage 2, and could follow the debugger with our assembly file! Cool, huh? Best of all, you can see the window dynamically update to display the output of your system.

### Single Step

The **s (Single Step)** command is used to walk through one instruction at a time:

```
< bochs:6> s
Next at t=934607
<0> [0x0000005a0] 0050:00a0 (ink. ctxt): cli            ; fa
< bochs:7> s
Next at t=934608
<0> [0x0000005a1] 0050:00a1 (ink. ctxt): xor AX, AX     ; 31c0
< bochs:8> _
```

### dump_cpu

This command displays the current value of all cpu registers, including RFLAGS, General Purpose, Test, Debug, Control, and Segment registers. It also includes GDTR, IDTR, LDTR, TR, and EIP.

### print_stack

This displays the current values of the stack. This is critical coinsidering that we use the stack very often.

### Conclusion

There are more commands then this, however these are the most usefull. Learning how to use the debugger is very important, espically in the early stages like we are in now.

## Direct Hardware Programming - Theory

This is where things start getting very hard in operating system development.

"Direct hardware programming" simply referrs to communicating directly (and controlling) individual chips. As long as these chips are programmable (In some way), we can control them.

In Tutorial 7, we took a very detailed look at how the system works. We also talked about how software ports work, port mapping, the IN and OUT instructions, and I gave a huge table with common port mappings on x86 architectures.

Remember that, whenever the processor recieves an **IN** or **OUT** instruction, It enables the **I/O Access Line** on the **Control Bus**, which, of course, is part of the **System Bus**, in the **Motherboards North Bridge**. Because the system bus is connected to both the **Memory Controller** and **I/O Controller**, both controllers listen for specific addresses and activated lines in the control bus. If the **I/O Access line** is set (Electricity runs through

it--which means it is active (1), The I/O controller takes the address.

The I/O Controller then gives the port address to every other device, and awaits a signal from a controller chip (meaning that it belongs to some device--so give whatever data to that device). If no controller chip responds, and the port address is set back, it is ignored.

This is how port mapping works. (Please see tutorial 7 for more detail.)

Also, remember that a single controller chip may be assigned a range of port addresses. Port addresses are assigned by the BIOS POST, even before the BIOS is loaded and executed. Why? Alot of devices need different types of information. Some ports may represent "registers", while others may be "data" or "ready" ports. It's ugly, I know. But it gets worse. On different systems, port addresses may vary widely. Because x86 architectures are backword compatible, basic devices (Such as keyboards and mice) are useually always the same address. More complex devices, however, may not be.

# Direct Hardware Programming and Controllers

To better understand how everything works, lets look at controllers. After all, we will be talking to them alot--espically in protected mode.

Many PCs are based off of the early **Intel 8042 Microcontroller chip**. This controller chip is either embedded as an IC (Intergrated Circuit) chip, or directly in the motherboard. It is useually locaed in the **South Bridge**.

This microcontroller communcates through a cord connecting to your keyboard, to another microcontroller chip in your keyboard.

When you press a key on the keyboard, It presses down on a rubber dome setting beneath the key itself. On the underside of the rubber dome is a conductive contact that, when pressed down, comes in contact with two conductive contacts on the keyboard circuit. Because of this, current can flow through. Each key is connected by a pair of electrical lines. As each signal changes (Depending on weather keys are pressed), a make code is generated (From the series of lines). This make code is sent to the microcontroller chip inside of the keyboard, and sent through the cord connecting to the computer hardware port. It is sent through as a series of on and off electrical pusles. Depending on the clock cycles, each pulse can be converted to a series of bits representing a bit pattern.

We are on the motherboard. This series of bits goes through the south bridge as electrical signals, all the way to the 8042 microcontroller. This microcontroller decodes the make code into a scan code, and stores it within an internal register. That is, our buffer. The internal registers can be an EEPROM chip, or simular, so we can electrically overwrite the data whenever we want.

When booting, the BIOS POST assigns each device (Through the I/O Controller) port addresses. It does this by quering the devices. In the useual case, the BIOS POST sets this internal register at port address 0x60. This means, **Whenever we refrence port 0x60, we are requesting to read from this internal register.**

You know the rest of the story reguarding port mapping, and IN/ OUT instructions, so lets read from that register:

```
in al, 0x60             ; get byte from 8042 microcontroller input register
```

As you can probably guess, **The 8042 Microcontroller is the Keyboard Controller**. By communicating with the various of registers with the chip, we can read input from the keyboard, map scan codes, even several other things: **Like enabling A20.**

You might be wondering why you have to communicate to the keyboard controller to enable A20. We will look at this next.

# Gate A20 - Theory

Finally we cover A20. I know, I know...Most of this tutorial so far covers other topics that are not directly related to A20. However, I wanted to start with the basics of direct hardware programming first before going into A20..as enabling A20 requires it, along with any microcontroller programming.

Enabling the A20 line may require programming the keyboard microcontroller. Because of this, I will cover a little bit about programming the keyboard controller but will not go into keyboard programming just yet.

## A little history

When IBM designed the **IBM PC AT** machines, it used their newer **Intel 80286 microprocessor**, which was not entirely compatible with previous x86 microprocessors when in real mode. The problem? The older x86 processors **did not** have address lines A20 through A31. They did not have an address bus that size yet. **Any programs that go beyond the first 1 MB would appear to wrap around.** While it worked back then, the 80286's address space required 32 address lines. However, if all 32 lines are accessable, we get the wrapping problem again.

To fix this problem, Intel put a logic gate on the 20th address line between the processor and system bus. This logic gate got named **Gate A20**, as it can be enabled and disabled. For older programs, it an be disabled for programs that rely on the wrap wround, and enabled for newer programs.

**When booting, the BIOS enables A20 when counting and testing memory, and then disables it again before giving our operating system control.**

There are alot of ways to enable A20. By enabling the A20 gate, we have access to all 32 lines on the address bus, and hence, can refrence 32 bit addresses, or up to 0xFFFFFFFF - 4 GB of memory.

The Gate A20 is an electronic OR gate that was originally connected to the P21 electrical line of the 8042 microcontroller (The keyboard controller). This gate is an output line that is treated as **Bit 1** of the output port data. We can send a command to recieve this data and even modify it. By setting this bit, and writing the output line data we can have the microcontroller set the OR gate thus enabling the A20 line. We can either do this ourself directly or indirectly. We will look more in the next section.

During bootup, the BIOS enables the A20 line to test the memory. After the memory test, the BIOS disables the A20 line to retain compatability with older processors. Because of this, by default, the A20 line is disabled for our operating system.

There can be several different ways to re-enable gate A20 depending on the motherboard configuation. Because of this, I will cover several different more common methods to enable A20.

Lets look at this next. ;)

# Gate A20 - Enabling

Remember that there are alot of different ways to enable A20. If you are wanting to just get your system working, all you need to do is use a method that works for you. If portability is a requirement, you may be required to use a mixture of methods.

# Method 1: System Control Port A

This is a very fast, yet less portable method of enabling the A20 address line.

Sone systems, including MCA and EISA we can control A20 from the system control port I/O 0x92. The exact details and functions of port 0x92 vary greatly by manufacturer. There are several bits that are commonly used though:

- **Bit 0** - Setting to 1 causes a fast reset (Used to switch back to real mode)
- **Bit 1** - 0: disable A20; 1: enable A20
- **Bit 2** - Manufacturer defined
- **Bit 3** - power on password bytes (CMOS bytes 0x38-0x3f or 0x36-0x3f). 0: accessible, 1: inaccessible
- **Bits 4-5** - Manufacturer defined
- **Bits 6-7** - 00: HDD activity LED off; any other value is "on"

Here is an example that enables A20 using this method:

```
mov al, 2 ; set bit 2 (enable a20)
out   0x92, al
```

Notice there is alot of other things we can do with this port:

```
mov al, 1 ; set bit 1 (fast reset)
out   0x92, al
```

This method seems to work with Bochs as well.

## Warning!

While this is one of the easier methods, I have seen this method conflict with some other hardware devices. It would normally cause the system to halt. If you want to use this method (and it works for you), I would stick with using it, but please keep this in mind.

## Other Ports...

I feel that I should mention that some systems allow the use of other I/O ports to enable and disable A20.

The most common of these are I/O port 0xEE. If I/O port 0xEE ("FAST A20 GATE") is enabled on these systems, reading from this port enables A20, and writing to it disables A20. A simular effect accors for port 0xEF ("FAST CPU RESET") as well for resetting the system.

Other systems may use different ports (ie; AT&T 6300+ needs a write of 0x90 to I/O port 0x3f20 to enable A20, and a write of 0 to disable A20). There are also rumours of systems that exist that use bit 2 of I/O port 0x65 or bit 0 of I/O port 0x1f8 for enabling and disabling A20 (0: disable, 1: enable).

As you can see, there are alot of headaches when it comes to working with A20. The only way to be sure is with your motherboard manufacturer.

## Method 2: Bios

Alot of Bios's make interrupts avilable for enabling and disabling A20.

### Bochs Support

It seems some versions of Bochs recognizes these methods but it may not be supported on some versions of Bochs.

### INT 0x15 Function 2400 - Disable A20

This function disables the A20 gate. It is very easy to use:

```
mov ax, 0x2400
int 0x15
```

Returns:

CF = clear if success
AH = 0
CF = set on error
AH = status (01=keyboard controller is in secure mode, 0x86=function not supported)

### INT 0x15 Function 2401 - Enable A20

This function enables the A20 gate.

```
mov ax, 0x2401
int 0x15
```

Returns:

CF = clear if success
AH = 0
CF = set on error
AH = status (01=keyboard controller is in secure mode, 0x86=function not supported)

### INT 0x15 Function 2402 - A20 Status

This function returns the current status of the A20 gate.

```
mov ax, 0x2402
int 0x15
```

Returns:

CF = clear if success
AH = status (01: keyboard controller is in secure mode; 0x86: function not supported)
AL = current state (00: disabled, 01: enabled)
CX = set to 0xffff is keyboard controller is no ready in 0xc000 read attempts
CF = set on error

### INT 0x15 Function 2403 - Query A20 support

This function is used to query the system for A20 support.

```
mov ax, 0x2403
int 0x15
```

Returns:

CF = clear if success
AH = status (01: keyboard controller is in secure mode; 0x86: function not supported)
BX = status.

BX contains a bit pattern:

- **Bit 0** - 1 if supported on keyboard controller
- **Bit 1** - 1 if supported on bit 1 of I/O port 0x92
- **Bits 2-14** - Reserved
- **Bit 15** - 1 if additional data is available.

## Method 3: Keyboard Controller

This is probably the most common method of enabling A20. Its quite easy, but requires some knowledge of programming the keyboard microcontroller. This will be the method I will be using as it seems it is also the most portable. Because this requires some knowledge of programming the keyboard microcontroller, we should look at that a little bit first.

This is also the reson why I wanted to cover hardware programming first. This will be our first glimps into direct hardware programming, and what it is all about. Don't worry, it's not too bad ;) It can get quite complex at times though :)

## 8043 Keyboard Controller - Port Mapping

Remember that -- in order for us to communicate with this controller, we must know what I/O ports the controller uses.

This controller has the following port mapping:

| Port Mapping | | |
|---|---|---|
| Port | Read/Write | Descripton |
| 0x60 | Read | Read Input Buffer |
| 0x60 | Write | Write Output Buffer |
| 0x64 | Read | Read Status Register |
| 0x64 | Write | Send Command to controller |

We send commands to this controller by writing the command byte to I/O Port 0x64. If the command accepts a parameter, this parameter is sent to port 0x60. Likewise, any results returned by the command may be read from port 0x60.

We must note that the keyboard controller itself is quite slow. Because our code will be executing faster then the keyboard controller, we must provide a way to wait for the controller to be ready before we continue on.

This is useually done by quering for the controllers status. If this seems confusing, don't worry--everything will be clear soon enough.

## 8043 Keyboard Controller Status Register

Okay, how do we get the status of the controller? Looking at the table above, we can see that we must read from I/O port 0x64. The value read from this register is an 8 bit value that follows a specific format. Here it is...

- **Bit 0: Output Buffer Status**
  - 0: Output buffer empty, dont read yet
  - 1: Output buffer full, please read me :)
- **Bit 1: Input Buffer Status**
  - 0: Input buffer empty, can be written
  - 1: Input buffer full, dont write yet
- **Bit 2:** System flag
  - 0: Set after power on reset
  - 1: Set after successfull completion of the keyboard controllers self-test (Basic Assurance Test, BAT)
- **Bit 3:** Command Data
  - 0: Last write to input buffer was data (via port 0x60)
  - 1: Last write to input buffer was a command (via port 0x64)
- **Bit 4:** Keyboard Locked
  - 0: Locked
  - 1: Not locked
- **Bit 5:** Auxiliary Output buffer full

    - PS/2 Systems:
        - 0: Determins if read from port 0x60 is valid If valid, 0=Keyboard data
        - 1: Mouse data, only if you can read from port 0x60
    - AT Systems:
        - 0: OK flag
        - 1: Timeout on transmission from keyboard controller to keyboard. **This may indicate no keyboard is present.**
- **Bit 6:** Timeout
    - 0: OK flag
    - 1: Timeout
    - PS/2:
        - General Timeout
    - AT:
        - Timeout on transmission from keyboard to keyboard controller. **Possibly parity error (In which case both bits 6 and 7 are set)**
- **Bit 7:** Parity error
    - 0: OK flag, no error
    - 1: Parity error with last byte

As you can see, there is alot going on here! The important bits are bolded above--they will tell us if the controllers output or input buffers are full or not.

Here is an example. Lets say we send a command to the controller. This is placed in the controllers input buffer. So, as long as this buffer is still full, we know our command is still being performed. Here is what our code might look like:

```
wait_input:
    in      al,0x64          ; read status register
    test    al,2       ; test bit 2 (Input buffer status)
    jnz     wait_input      ; jump if its not 0 (not empty) to continue waiting
```

We will be needing to do this for both the input and output buffers.

Now that we are able to wait for the controller, we must be able to actually tell the controller what we need it to do. This is done through command bytes. Lets take a look!

## 8043 Keyboard Controller Command Register

Looking back at the I/O port table, we can tell that we need to write to I/O Port 0x64 to send commands to the controller.

The keyboard controller has **alot** of commands. Because this is not a tutorial on keyboard programming, I will not list them all here. However, I will list the more important ones:

| Keyboard Controller Commands | |
|---|---|

| Keyboard Command | Descripton |
|---|---|
| 0x20 | Read Keyboard Controller Command Byte |
| 0x60 | Write Keyboard Controller Command Byte |
| 0xAA | Self Test |
| 0xAB | Interface Test |
| 0xAD | Disable Keyboard |
| 0xAE | Enable Keyboard |
| 0xC0 | Read Input Port |
| 0xD0 | Read Output Port |
| 0xD1 | Write Output Port |
| 0xDD | Enable A20 Address Line |
| 0xDF | Disable A20 Address Line |
| 0xE0 | Read Test Inputs |
| 0xFE | System Reset |
| Mouse Command | Descripton |
| 0xA7 | Disable Mouse Port |
| 0xA8 | Enable Mouse Port |
| 0xA9 | Test Mouse Port |
| 0xD4 | Write to mouse |

Again, please take note there are **alot** more commands then this. We will look at them all later, don't worry :)

## Method 3.1: Enabling A20 through keyboard controller

Notice the command bytes **0xDD** and **0xDF** in the above table. This is one way to enable A20 using the keyboard controller:

```
; Method 3.1: Enables A20 through keyboard controller
mov al, 0xdd    ; command 0xdd: enable a20
out 0x64, al    ; send command to controller
```

Not all keyboard controllers support this function. If it works, I would stick with it for its simplicity ;)

## Method 3.2: Enabling A20 through Output Port

Yet *another* method of enabling A20 is through the keyboard controllers output port. To do this, we need to use commands D0 and D1 to read and

write the output port (Please see the **Keyboard Controller Commands** table again for refrence.)

This method is a little bit more complex then the other methods, but it is not too bad. Basically, we can disable the keyboard and read the output port from the controller. The 8042 containes thee ports: One is input, the other is output. Oh right... The third is for testing. These "ports" are just the hardware pins on the microcontroller.

To keep things simple (And because this isnt a keyboard programming tutorial), we will just look at the output port for now.

Okay... read from the output port, simply send the...erm...read output port command (0xD0) to the controller: (Please see the keyboard controller commands table for refrence)

```
; read output port into al
mov     al,0xD0
out     0x64,al
```

Now we have gotten the output port data. Great, but that isnt very useful, is it? Well, actually the output port data follows..yet again..a specific bit format.

*Lets take a look...*

- **Bit 0: System Reset**
  - 0: Reset computer
  - 1: Normal operation
- **Bit 1: A20**
  - 0: Disabled
  - 1: Enabled
- **Bit 2-3:** Undefined
- **Bit 4:** Input buffer full
- **Bit 5:** Output Buffer Empty
- **Bit 6:** Keyboard Clock
  - 0: High-Z
  - 1: Pull Clock Low
- **Bit 6:** Keyboard Data
  - 0: High-Z
  - 1: Pull Data Low

Most of these bits we do not want to change. Setting bit 0 to 1 resets the computer; setting bit 1 enables gate A20. You should OR this value to set the bit to insure none of the other bits get touched. After setting the bit, just write the value back (Command byte 0xD1).

**The commands used to read and write the output port use the input and output buffers of the controller for its data.**

This means, if we read the output port, the data read will be in the controllers input buffer register. Looking back at the I/O port table, this means to get the data we read from I/O port 0x60.

Lets look at an example. During any read or write operation, we will want to wait for the controller to be ready. **wait_input** waits for the input buffer to be empty, while **wait_output** waits for the output buffer to be empty.

```
; send read output port command
mov     al,0xD0
out     0x64,al
call    wait_output

; read input buffer and store on stack. This is the data read from the output port
in      al,0x60
push    eax
call    wait_input

; send write output port command
mov     al,0xD1
out     0x64,al
call    wait_input

; pop the output port data from stack and set bit 1 (A20) to enable
pop     eax
or      al,2        // 2 = 10 binary
out     0x60,al         // write the data to the output port. This is done through the output buffer
```

Thats all there is to it! :) This method is a bit more complex then the other methods, but it is also the most portable.

## Cautions to look for

Because of it's emulation, most of these do not apply with Bochs, but to real hardware instead.

### Controller executes the wrong command

If the controller executes the wrong command, it will useually do something you don't want. Like, perhaps, read data from a port instead of write data), which may currupt your data. For example, using **in al, 0x61** instead of **in al, 0x60**, which will read from a different register in the keyboard microcontroller, instead of the status register (Port 0x60).

### Unkown Controller Command

Most controllers ignore commands it does not know, and just discards it (Clears it's command register, if any.)

Some controllers may malfunction, however. Please see the "Malfunction" section for more information.

### Controller Malfunctions

This happens rairly, but is possible. Two notible instances are both with the Pentium processor, including the infamous FDIV and foof bugs. The FDIV bug was an internal CPU design flaw, in which the FPU inside the processor gives incorrect results.

The foof problem is more series. When the processor is given the command bytes 0xf0 0x0f 0xc7 0xc8, which is an example of an **Hault and Catch Fire (HCF)** instruction. (An **Undocumented Instruction**). Most of these instructions may lock up the processor itself, forcing the user to hard reboot. Others may cause unusual side effects from the use of these instructions.

One should coinsider these problems that may happen. It does happen, and controllers are no exception. (Remember that we send instruction bytes to individual ports? For example, Port 0x64--The Command Register in the Keyboard Controller).

Most of these malfunctions can easily be coinsidered "Design Flaws" of the device, though.

### Physical Hardware damage

Although also rare, it is possible to infict hardware damage through software. An easy example is the floppy drive. You have to control the **floppy drive motor** directly through the **Floppy Drive Controller (FDC)**. Forgetting to send the command to stop the motor can cause the floppy drive to wear out and break. Be careful!

### Triple Fault

The microcontroller may signal the primary processor that there is a problem via the Control Bus, in which case the processor signals an exception, which will, of course, reboot the computer.

### Controller problems in Bochs

If there is a controller problem, Bochs will provoke a Triple fault, and log the information (The problem) into the log.

For example, if you try to send an unkown command (Such as 0) to the keyboard controller:

```
mov     al, 0x00        ; some random command
out     0x64, al        ; try to send command to controller
```

Bochs will provoke a triple fault, and log the information:

```
[KBD   ] unsupported io write to keyboard port 64, value = 0
```

"KBD" represents that the log was written by the keyboard controller device.

## Demo

All of the A20 code is in **A20.inc**. I wrote several different routines that uses different methods of enabling A20. So if one method fails, try using another method.

Do to the increase in complexity, I have decided to have this demo downloadable. The current **Stage2.asm** has not changed that much either.

Because the demo does not display anything new, there is not a new picture to display.

Download the latest demo (*.ZIP: 8KB) Here.

# Conclusion

Wow. Just, Wow. This tutorial is bigger then I originally expected.

We looked at alot of new concepts here. We also got experience with hardware programming. Remember: **This is the only way of communicating with hardware in protected mode!** Good bye interrupts. Good bye BIOS. Good bye everything--we are now completely on our own.

Right now, you can probably start appreciating Windows a little more :) After all, they all had to start at our level.

Don't worry if you do not understand everything yet--It is complicated, I know. When we get to our Kernel, we are going to have an entire tutorial dedicated to programming the keyboard microcontroller, and writing a driver for it. Cool?

The next tutorial will be much easier. We are going to put Protected Mode on hold for now, and go back to the real mode code. We will add the FAT12 loading code to load our kernel. Now that A20 is enabled, we can load it at 1MB!

Also, we will get some BIOS information, and anything else that comes to mind :) See you there.

Until next time,

~Mike
*BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System*

*Questions or comments? Feel free to Contact me.*

Would you like to contribute and help improve the articles? If so, please let me know!

Chapter 8 Home Chapter 10