Hello All! I'm coming today with a tutorial about IDE and how does it work, the tutorial is written on my own from scratch. any questions, comments or notes, i'm ready to hear from you.

before starting, notice that every thing here is being executed in Protected Mode, in a segment which is kernel-mode segment.

ok, lets start; but we should first Express what is an IDE. the IDE which we are talking about here is a keyword refers to Integrated Drive Electronics, Not that IDE which refers to (Integrated Development Environment).
The IDE is a part of the chipset which come with motherboard, we can cosider it as a device wich can be detected on PCI Bus. This Device manages IDE Drives which can be Hard-Disk Drives, Optical-Disk Drives [Like CD-ROMs, DVD-ROMs, Blue-Ray Drives]. IDE can allow even 4 drives to be connected to.
the drive may be:
1. Parallel AT-Attachment [PATA]: Like PATA HDDs
2. Parallel AT-Attachment Packet-Interface [PATAPI]: Like PATAPI ODDs.
3. Serial ATA [SATA]: Like SATA HDDs.
4. Serial ATAPI [SATAPI]: Like SATAPI ODDs.
We can ignore Tape Drives and ZIP Drives as they are obseleted.
The Way of accessing ATA Drives is one, means that the way of accessing PATA HDDs is the same of SATA HDDs. also the way of accessing PATAPI ODDs is the same of SATAPI ODDs.
for that, for IDE Device Driver, it is not required to know if a drive is Parallel or Serial, but it is important to know if it is ATA or ATAPI.

IDE Interface:
If you open your case and look at the mother board, we will see a port or two like this:

the white and green ports are IDE Ports, each port of them is called channel. so there is:
- Primary IDE Channel.
- Secondary IDE Channel.
These Ports allows only Parallel Drives to be connected to, means that it supports only PATA/PATAPI Drives.
Each Port can has a PATA cable connected to, it is like this:

Each PATA Cable can be connected with one master drive, or two drives [Master and Slave].
So we can have:
- Primary Master Drive.
- Primary Slave Drive.
- Secondary Master Drive.
- Secondary Slave Drive.
Each Drive May be: PATA or PATAPI.
But What about Serial IDE?
Almost many of modern motherboards have a Serial IDE which allows SATA and SATAPI Drives to be connected to.
Serial IDE Ports are 4:

Each Port is conducted with a Serial Cable:

So from the picture we can understand that only one drive can be connected to Serial IDE Port.
So each two ports make a channel, and also Serial IDE has:
- Primary Master Drive [Port1, or Port 2], also called [SATA1] in BIOS Setup Utility.
- Primary Slave Drive [Port 1 or Port 2], also called [SATA2] in BIOS Setup Utility.
- Secondary Master Drive [Port 3 or Port 4], also called [SATA3] in BIOS Setup Utility.
- Secondary Slave Drive [Port 3 or Port 4], also called [SATA4] in BIOS Setup Utility.

Please if you wanna support only the Parallel IDE, skip the part of [Detecting an IDE].

Detecting an IDE:
Each IDE appears as a device [in PCI World, it is called a function] on PCI Bus. If you don't know about PCI, please refer to http://wiki.osdev.org/PCI.
When you find a device on PCI, you can determine whether it is an IDE Device or not, this is determined according to **Class Code** and **Subclass code**.
If Class code is: 0x01 [Mass Storage Controller] and Subclass Code is: 0x01 [IDE], so this device is an IDE Device.
We know all that each PCI Device has 6 BARs, ok, only 5 BARs are used by IDE Device:
BAR0: Base Address of Primary Channel I/O Ports, if it is 0x0 or 0x1, this means [0x1F0].
BAR1: Base Address of Priamry Channel Control Ports, if it is 0x0 or 0x1, this means [0x3F4].
BAR2: Base Address of Secondary Channel I/O Ports, if it is 0x0 or 0x1, this means [0x170].
BAR3: Base Address of Secondary Channel Control Ports, if it is 0x0 or 0x1, this means [0x374].
BAR4: Bus Master IDE, this I/O Address refers to the base of I/O range consists of 16 ports, each 8 ports controls DMA on a channel.

IRQs are really a problem for IDEs, because the IDE uses IRQs 14 and 15, if it is a Parallel IDE.
If it is a Serial IDE, it uses another IRQ and only one IRQ, but how does we know the IRQs used by IDE? In Quafios it is quite easy:

Code:
```
outl((1<<31) | (bus<<16) | (device<<11) | (func<<8) | 8, 0xCF8); // Send the parameters.
if ((class = inl(0xCFC)>>16) != 0xFFFF) { // If there is exactly a device
   // Check if this device need an IRQ assignment:
   outl((1<<31) | (bus<<16) | (device<<11) | (func<<8) | 0x3C, 0xCF8);
   outb(0xFE, 0xCFC); // Change the IRQ field to 0xFE
   outl((1<<31) | (bus<<16) | (device<<11) | (func<<8) | 0x3C, 0xCF8); // Read the IRQ Field
Again.
   if ((inl(0xCFC) & 0xFF)==0xFE) {
      // This Device needs IRQ assignment.
   } else {
      // The Device doesn't use IRQs, check if this is an Parallel IDE:
      if (class == 0x01 && subclass == 0x01 && (ProgIF == 0x8A || ProgIF == 0x80)) {
         // This is a Parallel IDE Controller which use IRQ 14 and IRQ 15.
      }
   }
}
```

By this way, you can make a structure with PCI Devices, each device has IRQ0 and IRQ1, the both should have an initial value of 0xFF [No IRQ]. if we detect a PCI Device, if the Device needs an IRQ, we can change IRQ0. if the device on PCI doesn't need, but it is a Parallel IDE, we can edit IRQ0 to 14 and IRQ1 to 15.

When an IRQ is invoked, ISR should read the IRQ number from PIC, then it searches for the device which has this IRQ [in IRQ0 or IRQ1], and if the device is found, call the device driver to inform it that an IRQ is invoked.

Detecting IDE Drives
In Quafios, when an IDE Device is found, Quafios reserves a space in memory and copies Generic IDE Device Driver to this space, And then calls the device driver with a function number of 1. function number 1 is initialization, which is:

Code:
```
void ide_initialize(unsigned int BAR0, unsigned int BAR1, unsigned int BAR2, unsigned int
BAR3,
unsigned int BAR4) {
```

If you wanna support only the parallel IDE, you can put this command in kernel:
ide_initialize(0x1F0, 0x3F4, 0x170, 0x374, 0x000);
We can assume that BAR4 is 0x0 because we are not going to use it yet.

We will return to ide_initialize function which searches for drives connected to the IDE, before we are going into this function, we should write some functions which will help us a lot.
First We should write some Definitions:

Code:
```
#define    ATA_SR_BSY      0x80
#define    ATA_SR_DRDY      0x40
#define    ATA_SR_DF      0x20
#define    ATA_SR_DSC      0x10
#define    ATA_SR_DRQ      0x08
#define    ATA_SR_CORR      0x04
#define    ATA_SR_IDX      0x02
#define    ATA_SR_ERR      0x01
```

There is a port is called Command/Status Port, when it is read, you read the status of channel, the above bit maskes express these states.

Code:
```
#define    ATA_ER_BBK      0x80
#define    ATA_ER_UNC      0x40
#define    ATA_ER_MC      0x20
#define    ATA_ER_IDNF      0x10
#define    ATA_ER_MCR      0x08
#define    ATA_ER_ABRT      0x04
#define    ATA_ER_TK0NF   0x02
#define    ATA_ER_AMNF      0x01
```

There is a port is called Features/Error Port, if it is read, you are reading the errors of the last operation, the bit maskes above express these errors.

Code:
```
// ATA-Commands:
#define      ATA_CMD_READ_PIO          0x20
#define      ATA_CMD_READ_PIO_EXT      0x24
#define      ATA_CMD_READ_DMA          0xC8
#define      ATA_CMD_READ_DMA_EXT      0x25
#define      ATA_CMD_WRITE_PIO         0x30
#define      ATA_CMD_WRITE_PIO_EXT      0x34
#define      ATA_CMD_WRITE_DMA         0xCA
#define      ATA_CMD_WRITE_DMA_EXT      0x35
#define      ATA_CMD_CACHE_FLUSH      0xE7
#define      ATA_CMD_CACHE_FLUSH_EXT   0xEA
#define      ATA_CMD_PACKET            0xA0
#define       ATA_CMD_IDENTIFY_PACKET      0xA1
#define       ATA_CMD_IDENTIFY          0xEC
```

When you write to Command/Status Port, You are executing a command, which can be one of the commands above.

Code:
```
#define      ATAPI_CMD_READ      0xA8
#define      ATAPI_CMD_EJECT      0x1B
```

The Command above are for ATAPI Devices which will be understanded soon.

The Commands ATA_CMD_IDENTIFY_PACKET, and ATA_CMD_IDENTIFY, returns a buffer of 512 byte, the buffer is called Identification space, the following definitions are used to read information from the identification space.

Code:
```
#define    ATA_IDENT_DEVICETYPE   0
#define    ATA_IDENT_CYLINDERS   2
```

```
#define    ATA_IDENT_HEADS      6
#define    ATA_IDENT_SECTORS      12
#define    ATA_IDENT_SERIAL   20
#define    ATA_IDENT_MODEL      54
#define    ATA_IDENT_CAPABILITIES   98
#define    ATA_IDENT_FIELDVALID   106
#define    ATA_IDENT_MAX_LBA   120
#define   ATA_IDENT_COMMANDSETS   164
#define    ATA_IDENT_MAX_LBA_EXT   200
```

When you select a drive, you should specify if it is the master drive or the slave one:

Code:
```
#define      ATA_MASTER      0x00
#define      ATA_SLAVE      0x01
```

Code:
```
#define      IDE_ATA         0x00
#define      IDE_ATAPI      0x01
```

Code:
```
// ATA-ATAPI Task-File:
#define      ATA_REG_DATA      0x00
#define      ATA_REG_ERROR      0x01
#define      ATA_REG_FEATURES   0x01
#define      ATA_REG_SECCOUNT0   0x02
#define      ATA_REG_LBA0      0x03
#define      ATA_REG_LBA1      0x04
#define      ATA_REG_LBA2      0x05
#define      ATA_REG_HDDEVSEL   0x06
#define      ATA_REG_COMMAND      0x07
#define      ATA_REG_STATUS      0x07
#define      ATA_REG_SECCOUNT1   0x08
#define      ATA_REG_LBA3      0x09
#define      ATA_REG_LBA4      0x0A
#define      ATA_REG_LBA5      0x0B
#define      ATA_REG_CONTROL      0x0C
#define      ATA_REG_ALTSTATUS   0x0C
#define      ATA_REG_DEVADDRESS   0x0D
```

Task File is a range of ports [8 ports] which are used by primary channel [BAR0] or Secondary Channel [BAR2].

BAR0 + 0 is first port.
BAR0 + 1 is second port.
BAR0 + 3 is the third ... etc ...

if BAR0 is 0x1F0:
the Data Port of the Primary Channel is 0x1F0.
the Features/Error Port of the Priamry Channel is 0x1F1.
etc ...
the same with the secondary channel.

There is a port which is called "ALTSTATUS/CONTROL PORT", when is read, you read alternate status, when this port is written to, you are controlling a channel.

For the Primary Channel, ALTSTATUS/CONTROL Port is BAR1 + 2.
For the Secondary Channel, ALTSTATUS/CONTROL Port is BAR3 + 2.

We can know say that Each Channel has 13 Register, for a primary channel:

Data Register: BAR0[0]; // Read and Write

Error Register: BAR0[1]; // Read Only
Features Register: BAR0[1]; // Write Only
SECCOUNT0: BAR0[2]; // Read and Write
LBA0: BAR0[3]; // Read and Write
LBA1: BAR0[4]; // Read and Write
LBA2: BAR0[5]; // Read and Write
HDDEVSEL: BAR0[6]; // Read and Write, this port is used to select a drive in the channel.
Command Register: BAR0[7]; // Write Only.
Status Register: BAR0[7]; // Read Only.
Alternate Status Register: BAR1[2]; // Read Only.
Control Register: BAR1[2]; // Write Only.
DEVADDRESS: BAR1[2]; // I don't know what is the benefit from this register.

The map above is the same with the secondary channel, but it is using BAR2 and BAR3 instead of BAR0 and BAR1.

Code:
```
// Channels:
#define      ATA_PRIMARY      0x00
#define      ATA_SECONDARY      0x01

// Directions:
#define      ATA_READ      0x00
#define      ATA_WRITE      0x01
```

We have had defined all definitions needed by the driver, now lets move to an important part, we said that
BAR0 is the Base of I/O Ports used by Primary Channel.
BAR1 is the Base of I/O Ports which control Primary Channel.
BAR2 is the Base of I/O Ports used by Secondary Channel.
BAR3 is the Base of I/O Ports which control Secondary Channel.
BAR4 is the Base of 8 I/O Ports controls Primary Channel's Bus Master IDE [BMIDE].
BAR4 + 8 is the Base of 8 I/O Ports controls Secondary Channel's Bus Master IDE [BMIDE].

So we can make this global structure:

Code:
```
struct channel {
   unsigned short base;  // I/O Base.
   unsigned short ctrl;  // Control Base
   unsigned short bmide; // Bus Master IDE
   unsigned char  nIEN;  // nIEN (No Interrupt);
} channels[2];
```

We also need a buffer to read the identification space in it, we need a variable that indicates if an irq is invoked or not, and finally we need an array of 6 words [12 bytes] for ATAPI Drives:

Code:
```
unsigned char ide_buf[2048] = {0};
unsigned static char ide_irq_invoked = 0;
unsigned static char atapi_packet[12] = {0xA8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

We said the the IDE can contain up to 4 drives:

Code:
```
struct ide_device {
   unsigned char  reserved;    // 0 (Empty) or 1 (This Drive really exists).
   unsigned char  channel;     // 0 (Primary Channel) or 1 (Secondary Channel).
   unsigned char  drive;       // 0 (Master Drive) or 1 (Slave Drive).
```

```
   unsigned short type;        // 0: ATA, 1:ATAPI.
   unsigned short sign;       // Drive Signature
   unsigned short capabilities;// Features.
   unsigned int   commandsets; // Command Sets Supported.
   unsigned int   size;       // Size in Sectors.
   unsigned char  model[41];   // Model in string.
} ide_devices[4];
```

When we read a register in a channel, like STATUS Register, it is easy to execute:
ide_read(channel, ATA_REG_STATUS);

Code:
```
unsigned char ide_read(unsigned char channel, unsigned char reg) {
   unsigned char result;
   if   (reg > 0x07 && reg < 0x0C) ide_write(channel, ATA_REG_CONTROL, 0x80 |
channels[channel].nIEN);
   if   (reg < 0x08) result = inb(channels[channel].base  + reg - 0x00);
   else if   (reg < 0x0C) result = inb(channels[channel].base  + reg - 0x06);
   else if   (reg < 0x0E) result = inb(channels[channel].ctrl  + reg - 0x0A);
   else if   (reg < 0x16) result = inb(channels[channel].bmide + reg - 0x0E);
   if   (reg > 0x07 && reg < 0x0C) ide_write(channel, ATA_REG_CONTROL,
channels[channel].nIEN);
   return result;
}
```

And Also there is a function for writing to registers:

Code:
```
void ide_write(unsigned char channel, unsigned char reg, unsigned char data) {
   if   (reg > 0x07 && reg < 0x0C) ide_write(channel, ATA_REG_CONTROL, 0x80 |
channels[channel].nIEN);
   if   (reg < 0x08) outb(data, channels[channel].base  + reg - 0x00);
   else if   (reg < 0x0C) outb(data, channels[channel].base  + reg - 0x06);
   else if   (reg < 0x0E) outb(data, channels[channel].ctrl  + reg - 0x0A);
   else if   (reg < 0x16) outb(data, channels[channel].bmide + reg - 0x0E);
   if   (reg > 0x07 && reg < 0x0C) ide_write(channel, ATA_REG_CONTROL,
channels[channel].nIEN);
}
```

If We want to read the identification space, we should read Data Register as Double Word for 128 times. the first read is the first dword, the second read is the second dword, and so on. we can read the 128 dwords and copy them to our buffer.

Code:
```
void ide_read_buffer(unsigned char channel, unsigned char reg, unsigned int buffer, unsigned
int quads) {
   if   (reg > 0x07 && reg < 0x0C) ide_write(channel, ATA_REG_CONTROL, 0x80 |
channels[channel].nIEN);
   asm("pushw %es; movw %ds, %ax; movw %ax, %es");
   if   (reg < 0x08) insl(channels[channel].base  + reg - 0x00, buffer, quads);
   else if   (reg < 0x0C) insl(channels[channel].base  + reg - 0x06, buffer, quads);
   else if   (reg < 0x0E) insl(channels[channel].ctrl  + reg - 0x0A, buffer, quads);
   else if   (reg < 0x16) insl(channels[channel].bmide + reg - 0x0E, buffer, quads);
   asm("popw %es;");
   if   (reg > 0x07 && reg < 0x0C) ide_write(channel, ATA_REG_CONTROL,
channels[channel].nIEN);
}
```

When we send a command, we should wait for 400 nanosecond, then we should read Status Port, if Busy Bit is on, so we should read status port again, until Busy Bit is 0, in this case, we can read the results of the command. this operation is called "Polling", we can use IRQs instead of polling, and IRQs are suitable for Multi-Tasking Environments, but i think Polling is much faster than IRQs.

After Many Commands, if DF is set [Device Fault Bit], so there is a failure, and if DRQ is not set, so there is an error. if ERR bit is set, so there is an error which is described in Error Port.

Code:
```
unsigned char ide_polling(unsigned char channel, unsigned int advanced_check) {

   // (I) Delay 400 nanosecond for BSY to be set:
   // -------------------------------------------------
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.

   // (II) Wait for BSY to be cleared:
   // -------------------------------------------------
   while (ide_read(channel, ATA_REG_STATUS) & ATA_SR_BSY); // Wait for BSY to be zero.

   if (advanced_check) {

      unsigned char state = ide_read(channel, ATA_REG_STATUS); // Read Status Register.

      // (III) Check For Errors:
      // -------------------------------------------------
      if (state & ATA_SR_ERR) return 2; // Error.

      // (IV) Check If Device fault:
      // -------------------------------------------------
      if (state & ATA_SR_DF ) return 1; // Device Fault.

      // (V) Check DRQ:
      // -------------------------------------------------
      // BSY = 0; DF = 0; ERR = 0 so we should check for DRQ now.
      if (!(state & ATA_SR_DRQ)) return 3; // DRQ should be set

   }

   return 0; // No Error.

}
```

if there is an error, we have a functions which print errors on screen:

Code:
```
unsigned char ide_print_error(unsigned int drive, unsigned char err) {

   if (err == 0) return err;

   printk(" IDE:");
   if (err == 1) {printk("- Device Fault\n     "); err = 19;}
   else if (err == 2) {
      unsigned char st = ide_read(ide_devices[drive].channel, ATA_REG_ERROR);
      if (st & ATA_ER_AMNF)   {printk("- No Address Mark Found\n     ");   err = 7;}
      if (st & ATA_ER_TK0NF)   {printk("- No Media or Media Error\n     ");   err = 3;}
      if (st & ATA_ER_ABRT)   {printk("- Command Aborted\n     ");      err = 20;}
      if (st & ATA_ER_MCR)   {printk("- No Media or Media Error\n     ");   err = 3;}
      if (st & ATA_ER_IDNF)   {printk("- ID mark not Found\n     ");      err = 21;}
      if (st & ATA_ER_MC)   {printk("- No Media or Media Error\n     ");   err = 3;}
      if (st & ATA_ER_UNC)   {printk("- Uncorrectable Data Error\n     ");   err = 22;}
      if (st & ATA_ER_BBK)   {printk("- Bad Sectors\n     ");       err = 13;}
   } else  if (err == 3)           {printk("- Reads Nothing\n     "); err = 23;}
     else  if (err == 4)  {printk("- Write Protected\n     "); err = 8;}
   printk("- [%s %s] %s\n",
      (const char *[]){"Primary","Secondary"}[ide_devices[drive].channel],
      (const char *[]){"Master", "Slave"}[ide_devices[drive].drive],
      ide_devices[drive].model);

   return err;
}
```

Now lets return to the initialization function:

Code:
```
void ide_initialize(unsigned int BAR0, unsigned int BAR1, unsigned int BAR2, unsigned int
BAR3,
```

```
unsigned int BAR4) {

   int j, k, count = 0;

   // 1- Detect I/O Ports which interface IDE Controller:
   channels[ATA_PRIMARY  ].base  = (BAR0 &= 0xFFFFFFFC) + 0x1F0*(!BAR0);
   channels[ATA_PRIMARY  ].ctrl  = (BAR1 &= 0xFFFFFFFC) + 0x3F4*(!BAR1);
   channels[ATA_SECONDARY].base  = (BAR2 &= 0xFFFFFFFC) + 0x170*(!BAR2);
   channels[ATA_SECONDARY].ctrl  = (BAR3 &= 0xFFFFFFFC) + 0x374*(!BAR3);
   channels[ATA_PRIMARY  ].bmide = (BAR4 &= 0xFFFFFFFC) + 0; // Bus Master IDE
   channels[ATA_SECONDARY].bmide = (BAR4 &= 0xFFFFFFFC) + 8; // Bus Master IDE
```

Then We Should Disable IRQs in the both channels [This is temporary]:
This happens by setting bit 1 [nIEN] in Control Port:

Code:

```
   // 2- Disable IRQs:
   ide_write(ATA_PRIMARY  , ATA_REG_CONTROL, 2);
   ide_write(ATA_SECONDARY, ATA_REG_CONTROL, 2);
```

Now we need to check for drives connected to each channel, we will select the master drive of each channel, and send the command ATA_IDENTIFY (Which is supported by ATA Drives). if error, there is values returned in registers determines the type of Drive, if no drive, there will be strange values. Notice that bit4 in HDDEVSEL, if set to 1, we are selecting the slave drive, if set to 0, we are selecting the master drive.

Code:

```
   // 3- Detect ATA-ATAPI Devices:
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++) {

         unsigned char err = 0, type = IDE_ATA, status;
         ide_devices[count].reserved   = 0; // Assuming that no drive here.

         // (I) Select Drive:
         ide_write(i, ATA_REG_HDDEVSEL, 0xA0 | (j<<4)); // Select Drive.
         sleep(1); // Wait 1ms for drive select to work.

         // (II) Send ATA Identify Command:
         ide_write(i, ATA_REG_COMMAND, ATA_CMD_IDENTIFY);
         sleep(1); // This function should be implemented in your OS. which waits for 1 ms. it
is based on System Timer Device Driver.

         // (III) Polling:
         if (!(ide_read(i, ATA_REG_STATUS))) continue; // If Status = 0, No Device.

         while(1) {
            status = ide_read(i, ATA_REG_STATUS);
            if ( (status & ATA_SR_ERR)) {err = 1; break;} // If Err, Device is not ATA.
            if (!(status & ATA_SR_BSY) && (status & ATA_SR_DRQ)) break; // Everything is
right.
         }

         // (IV) Probe for ATAPI Devices:

         if (err) {
            unsigned char cl = ide_read(i,ATA_REG_LBA1);
            unsigned char ch = ide_read(i,ATA_REG_LBA2);

            if   (cl == 0x14 && ch ==0xEB) type = IDE_ATAPI;
            else if   (cl == 0x69 && ch ==0x96) type = IDE_ATAPI;
            else continue; // Unknown Type (And always not be a device).

            ide_write(i, ATA_REG_COMMAND, ATA_CMD_IDENTIFY_PACKET);
            sleep(1);
         }
```

```
         // (V) Read Identification Space of the Device:
         ide_read_buffer(i, ATA_REG_DATA, (unsigned int) ide_buf, 128);



         // (VI) Read Device Parameters:
         ide_devices[count].reserved   = 1;
         ide_devices[count].type      = type;
         ide_devices[count].channel   = i;
         ide_devices[count].drive   = j;
         ide_devices[count].sign      = ((unsigned short *) (ide_buf + ATA_IDENT_DEVICETYPE
 ))[0];
         ide_devices[count].capabilities = ((unsigned short *) (ide_buf +
ATA_IDENT_CAPABILITIES   ))[0];
         ide_devices[count].commandsets  = ((unsigned int   *) (ide_buf +




ATA_IDENT_COMMANDSETS   ))[0];

         // (VII) Get Size:
         if (ide_devices[count].commandsets & (1<<26)){
            // Device uses 48-Bit Addressing:
            ide_devices[count].size   = ((unsigned int   *) (ide_buf + ATA_IDENT_MAX_LBA_EXT
 ))[0];
            // Note that Quafios is 32-Bit Operating System, So last 2 Words are ignored.
         } else {
            // Device uses CHS or 28-bit Addressing:
            ide_devices[count].size   = ((unsigned int   *) (ide_buf + ATA_IDENT_MAX_LBA
 ))[0];
         }

         // (VIII) String indicates model of device (like Western Digital HDD and SONY DVD-
RW...):
         for(k = ATA_IDENT_MODEL; k < (ATA_IDENT_MODEL+40); k+=2) {
            ide_devices[count].model[k - ATA_IDENT_MODEL] = ide_buf[k+1];
            ide_devices[count].model[(k+1) - ATA_IDENT_MODEL] = ide_buf[k];}
         ide_devices[count].model[40] = 0; // Terminate String.

         count++;
      }

   // 4- Print Summary:
   for (i = 0; i < 4; i++)
      if (ide_devices[i].reserved == 1) {
         printk(" Found %s Drive %dGB - %s\n",
            (const char *[]){"ATA", "ATAPI"}[ide_devices[i].type],         /* Type */
            ide_devices[i].size/1024/1024/2,               /* Size */
            ide_devices[i].model);
      }
}
```

Read/Write From ATA Drive:

Now we are moving to a bit more advanced part, it is to read and write from/to an ATA Drive.
There is 3 ways of addressing a sector:
CHS (Cylinder-Head-Sector): an old way of addressing sectors in ATA drives, I think all ATA-Drives should support this way of addressing.
LBA28: Accessing a sector by its LBA Address. but the address should be 28-bit long. i think all ATA-Drives should support this way of addressing
the problem of LBA28 Addressing is that it allows only to access 128GB from the ATA-Disk, so if ATA-Disk is more than 128GB, it should support LBA48 Feature Set.
LBA48: Accessing a sector by its LBA Address. but the address should be 48-bit long. as we use integers in GCC, so our maximum address in this tutorial is 32-bit long, which allows accessing an ATA-Drive up to 2TB.

So We can conclude an algorithm to determine which type of Addressing we are going to use:
If (Drive doesn't Support LBA)
// Use CHS.
else (if the LBA Sector Address > 0x0FFFFFFF)
// The Sector We are going to read is above 128GB Boundary, Use LBA48.
else // Use LBA28.

Reading the buffer may be done by polling or DMA.
PIO: After sending the command [Read or Write Sectors], we read Data Port [as words], or write to Data Port [as words]. this is the same way of reading identification space.
DMA: After sending the command, you should wait for an IRQ, while you are waiting, Buffer is written directly to memory automatically.
We are going to use PIO in our tutorial as it isn't going to be complix, and i want to be far from IRQs as they are very slower than Polling after PIO.

We can conclude also this table:

Code:

```
   /* ATA/ATAPI Read/Write Modes:
    * ++++++++++++++++++++++++++++++
    *  Addressing Modes:
    *  ================
    *   - LBA28 Mode.          (+)
    *   - LBA48 Mode.          (+)
    *   - CHS.             (+)
    *  Reading Modes:
    *  ================
    *   - PIO Modes (0 : 6)       (+) // Slower than DMA, but not a problem.
    *   - Single Word DMA Modes (0, 1, 2).
    *   - Double Word DMA Modes (0, 1, 2).
    *   - Ultra DMA Modes (0 : 6).
    *  Polling Modes:
    *  ================
    *   - IRQs
    *   - Polling Status         (+) // Suitable for Singletasking
    */
```

there is something needed to be expressed here, I have told before that Task-File is like that:
Register 0: [Word] Data Register. [Readable & Writable].
Register 1: [Byte] Error Register. [Readable].
Register 1: [Byte] Features Register. [Writable].
Register 2: [Byte] SECCOUNT0 Register. [Readable & Writable].
Register 3: [Byte] LBA0 Register. [Readable & Writable].
Register 4: [Byte] LBA1 Register. [Readable & Writable].
Register 5: [Byte] LBA2 Register. [Readable & Writable].
Register 6: [Byte] HDDEVSEL Register. [Readable & Writable].
Register 7: [Byte] Command Register. [Writable].
Register 7: [Byte] Status Register. [Readable].

So each one of Registers from 2 to 5 should be 8-bits long. but really each one of them is 16-bit long.

Register 2: [Bits 0-7] SECCOUNT0, [Bits 8-15] SECOUNT1
Register 3: [Bits 0-7] LBA0, [Bits 8-15] LBA3
Register 4: [Bits 0-7] LBA1, [Bits 8-15] LBA4
Register 5: [Bits 0-7] LBA2, [Bits 8-15] LBA5

the word [(SECCOUNT1<<8) | SECCOUNT0] expresses number of sectors which can be read when you access by LBA48.
when you access in CHS or LBA28, SECCOUNT0 only expresses number of sectors.

LBA0 is Bits[0-7] of LBA Address when you read in LBA28 or LBA48, it can be sector number of CHS.
LBA1 is Bits[8-15] of LBA Address when you read in LBA28 or LBA48, it can be low 8 bits of cylinder number

of CHS.
LBA2 is Bits[16-23] of LBA Address when you read in LBA28 or LBA48, it can be high 8 bits of cylinder number of CHS.
LBA3 is Bits[24-31] of LBA Address when you read in LBA48.
LBA4 is Bits[32-39] of LBA Address when you read in LBA48.
LBA5 is Bits[40-47] of LBA Address when you read in LBA48.

notice that according to that, LBA0,1,2 registers [8-bits + 8-bits + 8-bits] are 24-bit long, which is not enough for LBA28, so the higher 4-bits can be written to the lower 4-bits of HDDEVSEL Register.

Also notice that if we set bit 6 of this register, we are going to use LBA, if not, we are going to use CHS. notice that there is a mode which is called extended CHS, but i don't wanna be exposed to that.

Lets go into the code:

Code:
```
unsigned char ide_ata_access(   unsigned char direction, unsigned char drive, unsigned int
lba, unsigned char numsects, unsigned short selector, unsigned int edi) {
```

This Function reads/writes sectors from ATA-Drive. if (direction = 0) so we are reading, else we are writing.
drive, is drive number which can be from 0 to 3.
lba, is the LBA Address which allows us to access disks up to 2TB.
numsects, number of sectors to be read, it is a char, as reading more than 256 sector immediately may cause the OS to hang. notice that if numsects = 0, controller will know that we want 256 sectors.
selector, segment selector to read from, or write to.
edi, offset in the segment.

Code:
```
   unsigned char lba_mode /* 0: CHS, 1:LBA28, 2: LBA48 */, dma /* 0: No DMA, 1: DMA */, cmd;
   unsigned char lba_io[6];
   unsigned int  channel      = ide_devices[drive].channel; // Read the Channel.
   unsigned int  slavebit      = ide_devices[drive].drive; // Read the Drive [Master/Slave]
   unsigned int  bus      = channels[channel].base; // The Bus Base, like [0x1F0] which is
also data port.
   unsigned int  words      = 256; // Approximatly all ATA-Drives has sector-size of 512-byte.
   unsigned short cyl, i; unsigned char head, sect, err;
```

We don't need IRQs, so we should disable it to disallow problems to happen, we said before that bit 1 of Control Register (Which is called nIEN bit), if it is set, so no IRQs will be invoked from this channel, either from Master Drive or from Slave Drive.

Code:
```
ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN = (ide_irq_invoked = 0x0) + 0x02);
```

Now lets set the parameters:

Code:
```
   // (I) Select one from LBA28, LBA48 or CHS;
   if (lba >= 0x10000000) { // Sure Drive should support LBA in this case, or you are giving a
wrong LBA.
      // LBA48:
      lba_mode  = 2;
      lba_io[0] = (lba & 0x000000FF)>> 0;
      lba_io[1] = (lba & 0x0000FF00)>> 8;
      lba_io[2] = (lba & 0x00FF0000)>>16;
      lba_io[3] = (lba & 0xFF000000)>>24;
      lba_io[4] = 0; // We said that we lba is integer, so 32-bit are enough to access 2TB.
      lba_io[5] = 0; // We said that we lba is integer, so 32-bit are enough to access 2TB.
      head      = 0; // Lower 4-bits of HDDEVSEL are not used here.
   } else if (ide_devices[drive].capabilities & 0x200)  { // Drive supports LBA?
      // LBA28:
```

```
      lba_mode  = 1;
      lba_io[0] = (lba & 0x00000FF)>> 0;
      lba_io[1] = (lba & 0x000FF00)>> 8;
      lba_io[2] = (lba & 0x0FF0000)>>16;
      lba_io[3] = 0; // These Registers are not used here.
      lba_io[4] = 0; // These Registers are not used here.
      lba_io[5] = 0; // These Registers are not used here.
      head      = (lba & 0xF000000)>>24;
   } else {
      // CHS:
      lba_mode  = 0;
      sect      = (lba % 63) + 1;
      cyl       = (lba + 1  - sect)/(16*63);
      lba_io[0] = sect;
      lba_io[1] = (cyl>>0) & 0xFF;
      lba_io[2] = (cyl>>8) & 0xFF;
      lba_io[3] = 0;
      lba_io[4] = 0;
      lba_io[5] = 0;
      head      = (lba + 1  - sect)%(16*63)/(63); // Head number is written to HDDEVSEL lower
4-bits.
   }
```

Now we are going to choose the way of reading the buffer [PIO or DMA]:

Code:

```
   // (II) See if Drive Supports DMA or not;
   dma = 0; // Supports or doesn't, we don't support !!!
```

Lets Poll the status port if the channel is busy:

Code:

```
   // (III) Wait if the drive is busy;
   while (ide_read(channel, ATA_REG_STATUS) & ATA_SR_BSY); // Wait if Busy.
```

HDDDEVSEL Register now looks like this:

Bits 0-3: Head Number for CHS.
Bit 4: Slave Bit. (0: Selecting Master Drive, 1: Selecting Slave Drive).
Bit 5: Obselete and isn't used, but should be set.
Bit 6: LBA (0: CHS, 1: LBA).
Bit 7: Obselete and isn't used, but should be set.

Lets write all these information to the register, while the obselete bits are set (0xA0):

Code:

```
   // (IV) Select Drive from the controller;
   if (lba_mode == 0) ide_write(channel, ATA_REG_HDDEVSEL, 0xA0 | (slavebit<<4) | head);   //
Select Drive CHS.
   else         ide_write(channel, ATA_REG_HDDEVSEL, 0xE0 | (slavebit<<4) | head);   // Select
Drive LBA.
```

Let's write the parameters to registers:

Code:

```
   // (V) Write Parameters;
   if (lba_mode == 2) {
      ide_write(channel, ATA_REG_SECCOUNT1,   0);
      ide_write(channel, ATA_REG_LBA3,   lba_io[3]);
      ide_write(channel, ATA_REG_LBA4,   lba_io[4]);
      ide_write(channel, ATA_REG_LBA5,   lba_io[5]);
   }
   ide_write(channel, ATA_REG_SECCOUNT0,   numsects);
   ide_write(channel, ATA_REG_LBA0,   lba_io[0]);
```

```
   ide_write(channel, ATA_REG_LBA1,   lba_io[1]);
   ide_write(channel, ATA_REG_LBA2,   lba_io[2]);
```

By this way, if you are using LBA48 and want to write to LBA0 Register [Register 3 in Task-File], and want to write to LBA3 Register [Register 3 also in Task-File], you should write LBA3 to Register 3, then write LBA0 to Register 3. ide_write function makes it quite simple, refer to the function and you will full-understand the code.

Now, we have a great set of commands described in ATA/ATAPI-8 Specification, we should choose the suitable command to execute:

Code:
```
   // (VI) Select the command and send it;
   // Routine that is followed:
   // If ( DMA & LBA48)   DO_DMA_EXT;
   // If ( DMA & LBA28)   DO_DMA_LBA;
   // If ( DMA & LBA28)   DO_DMA_CHS;
   // If (!DMA & LBA48)   DO_PIO_EXT;
   // If (!DMA & LBA28)   DO_PIO_LBA;
   // If (!DMA & !LBA#)   DO_PIO_CHS;
```

There isn't a command for Doing in CHS with DMA.

Code:
```
   if (lba_mode == 0 && dma == 0 && direction == 0) cmd = ATA_CMD_READ_PIO;
   if (lba_mode == 1 && dma == 0 && direction == 0) cmd = ATA_CMD_READ_PIO;
   if (lba_mode == 2 && dma == 0 && direction == 0) cmd = ATA_CMD_READ_PIO_EXT;
   if (lba_mode == 0 && dma == 1 && direction == 0) cmd = ATA_CMD_READ_DMA;
   if (lba_mode == 1 && dma == 1 && direction == 0) cmd = ATA_CMD_READ_DMA;
   if (lba_mode == 2 && dma == 1 && direction == 0) cmd = ATA_CMD_READ_DMA_EXT;
   if (lba_mode == 0 && dma == 0 && direction == 1) cmd = ATA_CMD_WRITE_PIO;
   if (lba_mode == 1 && dma == 0 && direction == 1) cmd = ATA_CMD_WRITE_PIO;
   if (lba_mode == 2 && dma == 0 && direction == 1) cmd = ATA_CMD_WRITE_PIO_EXT;
   if (lba_mode == 0 && dma == 1 && direction == 1) cmd = ATA_CMD_WRITE_DMA;
   if (lba_mode == 1 && dma == 1 && direction == 1) cmd = ATA_CMD_WRITE_DMA;
   if (lba_mode == 2 && dma == 1 && direction == 1) cmd = ATA_CMD_WRITE_DMA_EXT;
   ide_write(channel, ATA_REG_COMMAND, cmd);               // Send the Command.
```

This Command "ATA_CMD_READ_PIO" is right for rading in LBA28 or CHS, and controller refers to bit 6 of HDDEVSEL Register to know the mode of reading (LBA or CHS).

After sending the command, we should poll, then we read/write a sector then we should poll, then we read/write a sector, until we read/write all sectors needed, if an error is happened, we the function will return a specific error code.

notice that after writing, we should execute the CACHE FLUSH Command, and we should poll after it, but without checking for errors.

Code:
```
   if (dma)
      if (direction == 0);
      // DMA Read.
      else; // DMA Write.
   else
      if (direction == 0)
      // PIO Read.
      for (i = 0; i < numsects; i++) {
         if (err = ide_polling(channel, 1)) return err; // Polling, then set error and exit if
there is.
         asm("pushw %es");
         asm("mov %%ax, %%es"::"a"(selector));
         asm("rep insw"::"c"(words), "d"(bus), "D"(edi)); // Receive Data.
         asm("popw %es");
         edi += (words*2);
      } else {
      // PIO Write.
      for (i = 0; i < numsects; i++) {
```

```
         ide_polling(channel, 0); // Polling.
         asm("pushw %ds");
         asm("mov %%ax, %%ds"::"a"(selector));
         asm("rep outsw"::"c"(words), "d"(bus), "S"(edi)); // Send Data
         asm("popw %ds");
         edi += (words*2);
         }
      ide_write(channel, ATA_REG_COMMAND, (char []) {   ATA_CMD_CACHE_FLUSH,
                        ATA_CMD_CACHE_FLUSH,
                        ATA_CMD_CACHE_FLUSH_EXT}[lba_mode]);
      ide_polling(channel, 0); // Polling.
      }

   return 0; // Easy, ... Isn't it?
}
```

Read From ATAPI Drive:

Let's move to a part which is quite easier, it is to read from ATAPI Drive, i will not make the function write to ATAPI Drive, because the write Operation is very complix and it should done by third-party tools (like Nero in Windows, and Brasero in Linux).

ATAPI Drive is different from ATA Drives, as it doesn't use ATA Commands, but it use the SCSI-Command-Set. Parameters are sent into a Packet, so it is Called: ATA-Packet Interface [ATAPI].

Notice also that ATAPI drives should always use IRQs, you can't disable them, so we should create a function which waits for an IRQ to be caused:

Code:
```
void ide_wait_irq() {
   while (!ide_irq_invoked);
   ide_irq_invoked = 0;
}
```

when an IRQ happens, the following function should be executed by ISR:

Code:
```
void ide_irq() {
   ide_irq_invoked = 1;
}
```

by this way, ide_wait_irq() will go into a while loop, which waits for the variable ide_irq_invoked to be set, then it reclears it.

Code:
```
unsigned char ide_atapi_read(unsigned char drive, unsigned int lba, unsigned char numsects,
          unsigned short selector, unsigned int edi) {
```

drive, is the drive number, which is from 0 to 3.
lba, the lba address.
numsects, number of sectors, it should always be 1, and if you wanna read more than one sector, re-execute this fucntion with updated LBA address.
selector, Segment Selector.
edi, offset in the selector.

let's read the parameters of the drive:

Code:
```
   unsigned int   channel      = ide_devices[drive].channel;
   unsigned int   slavebit      = ide_devices[drive].drive;
   unsigned int   bus      = channels[channel].base;
```

```
   unsigned int   words      = 2048 / 2; // Sector Size in Words, Almost All ATAPI Drives has
a sector size of 2048 bytes.
   unsigned char  err; int i;
```

we need IRQs:

Code:

```
   // Enable IRQs:
   ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN = ide_irq_invoked = 0x0);
```

Let's setup the SCSI Packet, Which is 6-Words long [12-Bytes]:

Code:

```
   // (I): Setup SCSI Packet:
   // ------------------------------------------------------------------
   atapi_packet[ 0] = ATAPI_CMD_READ;
   atapi_packet[ 1] = 0x0;
   atapi_packet[ 2] = (lba>>24) & 0xFF;
   atapi_packet[ 3] = (lba>>16) & 0xFF;
   atapi_packet[ 4] = (lba>> 8) & 0xFF;
   atapi_packet[ 5] = (lba>> 0) & 0xFF;
   atapi_packet[ 6] = 0x0;
   atapi_packet[ 7] = 0x0;
   atapi_packet[ 8] = 0x0;
   atapi_packet[ 9] = numsects;
   atapi_packet[10] = 0x0;
   atapi_packet[11] = 0x0;
```

Now we should select the drive:

Code:

```
   // (II): Select the Drive:
   // ------------------------------------------------------------------
   ide_write(channel, ATA_REG_HDDEVSEL, slavebit<<4);
```

400 nanosecond after this select is a good idea:

Code:

```
   // (III): Delay 400 nanosecond for select to complete:
   // ------------------------------------------------------------------
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
   ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
```

Code:

```
   // (IV): Inform the Controller that we use PIO mode:
   // ------------------------------------------------------------------
   ide_write(channel, ATA_REG_FEATURES, 0);         // PIO mode.
```

Controller wants to know what do we think of the size of buffer!, we will allow him to know that:

Code:

```
   // (V): Tell the Controller the size of buffer:
   // ------------------------------------------------------------------
   ide_write(channel, ATA_REG_LBA1, (words * 2) & 0xFF);   // Lower Byte of Sector Size.
   ide_write(channel, ATA_REG_LBA2, (words * 2)>>8);   // Upper Byte of Sector Size.
```

Now we want to send the packet, we should first send the command "Packet":

Code:

```
   // (VI): Send the Packet Command:
   // -------------------------------------------------------------
   ide_write(channel, ATA_REG_COMMAND, ATA_CMD_PACKET);      // Send the Command.
```

Code:
```
   // (VII): Waiting for the driver to finish or invoke an error:
   // -------------------------------------------------------------
   if (err = ide_polling(channel, 1)) return err;         // Polling and return if error.
```

Code:
```
   // (VIII): Sending the packet data:
   // -------------------------------------------------------------
   asm("rep   outsw"::"c"(6), "d"(bus), "S"(atapi_packet));   // Send Packet Data
```

here we cannot Poll, we should wait for an IRQ, then read the sectors. these two operations should be repeated as the number of sectors, but we are said before that numsects should be 1. But I have put a for loop, i don't know why.

Code:
```
   // (IX): Recieving Data:
   // -------------------------------------------------------------
   for (i = 0; i < numsects; i++) {
      ide_wait_irq();                  // Wait for an IRQ.
      if (err = ide_polling(channel, 1)) return err;      // Polling and return if error.
      asm("pushw %es");
      asm("mov %%ax, %%es"::"a"(selector));
      asm("rep insw"::"c"(words), "d"(bus), "D"(edi));// Receive Data.
      asm("popw %es");
      edi += (words*2);
   }
```

Now we should wait for an IRQ and Poll for Busy and DRQ bits to be clear:

Code:
```
   // (X): Waiting for an IRQ:
   // -------------------------------------------------------------
   ide_wait_irq();

   // (XI): Waiting for BSY & DRQ to clear:
   // -------------------------------------------------------------
   while (ide_read(channel, ATA_REG_STATUS) & (ATA_SR_BSY | ATA_SR_DRQ));

   return 0; // Easy, ... Isn't it?
}
```

Standard Function For Reading from ATA/ATAPI Drive:

Code:
```
void ide_read_sectors(unsigned char drive, unsigned char numsects, unsigned int lba, unsigned
short es, unsigned int edi) {

   // 1: Check if the drive presents:
   // ==================================
   if (drive > 3 || ide_devices[drive].reserved == 0) package[0] = 0x1;      // Drive Not
Found!

   // 2: Check if inputs are valid:
   // ==================================
   else if (((lba + numsects) > ide_devices[drive].size) && (ide_devices[drive].type ==
IDE_ATA))
      package[0] = 0x2;                     // Seeking to invalid position.
```

```
   // 3: Read in PIO Mode through Polling & IRQs:
   // ============================================
   else {
      unsigned char err;
      if (ide_devices[drive].type == IDE_ATA)
         err = ide_ata_access(ATA_READ, drive, lba, numsects, es, edi);
      else if (ide_devices[drive].type == IDE_ATAPI)
         for (i = 0; i < numsects; i++)
            err = ide_atapi_read(drive, lba + i, 1, es, edi + (i*2048));
      package[0] = ide_print_error(drive, err);
   }
}
// package[0] is an entry of array, this entry specifies the Error Code, you can replace that.
```

Standard Function to write to ATA Drive:

Code:

```
void ide_write_sectors(unsigned char drive, unsigned char numsects, unsigned int lba, unsigned
short es, unsigned int edi) {

   // 1: Check if the drive presents:
   // ==================================
   if (drive > 3 || ide_devices[drive].reserved == 0) package[0] = 0x1;      // Drive Not
Found!
   // 2: Check if inputs are valid:
   // ==================================
   else if (((lba + numsects) > ide_devices[drive].size) && (ide_devices[drive].type ==
IDE_ATA))
      package[0] = 0x2;                     // Seeking to invalid position.
   // 3: Read in PIO Mode through Polling & IRQs:
   // ============================================
   else {
      unsigned char err;
      if (ide_devices[drive].type == IDE_ATA)
         err = ide_ata_access(ATA_WRITE, drive, lba, numsects, es, edi);
      else if (ide_devices[drive].type == IDE_ATAPI)
         err = 4; // Write-Protected.
      package[0] = ide_print_error(drive, err);
   }
}
```

Standard Function to eject ATAPI Drive:

Code:

```
void ide_atapi_eject(unsigned char drive) {
   unsigned int   channel      = ide_devices[drive].channel;
   unsigned int   slavebit      = ide_devices[drive].drive;
   unsigned int   bus      = channels[channel].base;
   unsigned int   words      = 2048 / 2;               // Sector Size in Words.
   unsigned char  err = 0;
   ide_irq_invoked = 0;

   // 1: Check if the drive presents:
   // ==================================
   if (drive > 3 || ide_devices[drive].reserved == 0) package[0] = 0x1;      // Drive Not
Found!
   // 2: Check if drive isn't ATAPI:
   // ==================================
   else if (ide_devices[drive].type == IDE_ATA) package[0] = 20;         // Command Aborted.
   // 3: Eject ATAPI Driver:
   // ============================================
   else {
      // Enable IRQs:
      ide_write(channel, ATA_REG_CONTROL, channels[channel].nIEN = ide_irq_invoked = 0x0);

      // (I): Setup SCSI Packet:
      // ------------------------------------------------------------------
```

```
      atapi_packet[ 0] = ATAPI_CMD_EJECT;
      atapi_packet[ 1] = 0x00;
      atapi_packet[ 2] = 0x00;
      atapi_packet[ 3] = 0x00;
      atapi_packet[ 4] = 0x02;
      atapi_packet[ 5] = 0x00;
      atapi_packet[ 6] = 0x00;
      atapi_packet[ 7] = 0x00;
      atapi_packet[ 8] = 0x00;
      atapi_packet[ 9] = 0x00;
      atapi_packet[10] = 0x00;
      atapi_packet[11] = 0x00;

      // (II): Select the Drive:
      // ------------------------------------------------------------------
      ide_write(channel, ATA_REG_HDDEVSEL, slavebit<<4);

      // (III): Delay 400 nanosecond for select to complete:
      // ------------------------------------------------------------------
      ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
      ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
      ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.
      ide_read(channel, ATA_REG_ALTSTATUS); // Reading Alternate Status Port wastes 100ns.

      // (IV): Send the Packet Command:
      // ------------------------------------------------------------------
      ide_write(channel, ATA_REG_COMMAND, ATA_CMD_PACKET);      // Send the Command.

      // (V): Waiting for the driver to finish or invoke an error:
      // ------------------------------------------------------------------
      if (err = ide_polling(channel, 1));            // Polling and stop if error.

      // (VI): Sending the packet data:
      // ------------------------------------------------------------------
      else {
         asm("rep   outsw"::"c"(6), "d"(bus), "S"(atapi_packet));// Send Packet Data
         ide_wait_irq();                  // Wait for an IRQ.
         err = ide_polling(channel, 1);            // Polling and get error code.
         if (err == 3) err = 0; // DRQ is not needed here.
      }
      package[0] = ide_print_error(drive, err); // Return;

   }
}
```

Now you can have your ODD is ejected:

I hope the tutorial to be easy for you and to be understanded, any one wants to write an IDE Drive, he can use the code in the tutorial.
Any comments, questions, or any thing, i'm ready to hear from you.
and thanx for anyone interested in reading the tutorial.

_____
Quafios, my hobby OS.
My personal website.

Last edited by iocoder on Sun Nov 08, 2009 8:38 am, edited 3 times in total.