The MP3 Player Project

Alex Olson

12/03/2002

1 Objective

To design and build an mp3 player capable of playing files from an IDE harddrive. Modern harddrives can store far more data compared to currently available CompactFlash memory cards. Although there are a few commercially available players that play from a harddrive, I wanted to make my own. I wanted to be able to add whatever features I saw fit.

2 Theory & Design

2.1 MP3 Format

Storing music in the MP3 format is useful because this format records high-fidelity audio in about 1/10th the space normally required for uncompressed audio. The algorithm for decoding and playing mp3s is fairly complex. There are a few chips currently available than can convert a MP3 data stream into analog audio. Most of them consist of streaming data to them in a serial manner, and they output analog audio (for driving headphones). I chose the VLSI 1001k, made by VLSI Solutions. It has two synchronous serial buses: one for data, one for control. It also has a 2Kbyte buffer and a *Data Request* line that it asserts when it can accept at least of 32 bytes of data. To play a song, all that is needed is to first reset the device (by sending a certain sequence down the control bus, and send data in 32 byte blocks (as long as the request line is high at the beginning of each block). There is a *BSYNC* signal that should be raised before the first bit of every byte. The purpose of the *BSYNC* signal is that ensures proper bit alignment is maintained. There are a few registers that can be accessed with the control bus. They provide info such as time elapsed, bit-rate information, mono/stereo, etc. Only the time elapsed is used in my project.

2.2 IDE Harddrives

IDE harddrives were designed to have a simple, cheap interface. They function completely asynchronously. All required logic for physically controlling the drive is built-in on each harddrive. The interface merely handles the data going in/out. There are a set of 8-bit registers such as *Sector Number*, *Sector Count*, *Command*, *Status*, *Error*, *Cylinder High*, *Cylinder Low*, *Drive/Head* etc. There are about 14 registers that are divided into two blocks: the *Command Block* and the *Control block*. There are two enable lines for the harddrive, one for each register block. It is not valid to enable both register lines simultaneously. All registers listed are in the *Command Block*; the other registers in the *Control Block* are rarely used and are not necessary for basic operations. There is also one *Data* register that is 16 bits wide in the *Command Block*. Basically, all transactions are initiated by first checking the *Status* register to see if the harddrive is not busy. To read or write to the harddrive's registers, the harddrive is first selected and *then* either the DIOR or DIOW line is asserted to indicate a read or write, respectively. This is somewhat different than many other

devices, including the LCD display. Bit 7 in the *Status* register is set to one when the harddrive is busy. While the harddrive is busy, no other register (other than the status register) should be accessed. Then the *Sector Number*, *Sector Count*, *Cylinder High*, *Cylinder Low* registers are written with appropriate values. Next, the *Command* register is written with a particular command, such as the *Read Sector* command. The harddrive will then go into the busy state (and set its *BUSY* flag in its status register). When the host sees the harddrive is no longer busy, it can read the data register to get one word. For sector transfers, a sector is defined as 512 bytes for all harddrives. By reading the *Data* register 256 times, the contents of an entire sector can be transferred. There are two modes for addressing locations on the harddrive, *CHS*, and *LBA*. *CHS* stands for Cylinders/Heads/Sectors. It was the original way of addressing sectors and is hardware-dependent. (The 1000th sector on a harddrive could have any CHS address, based on the size and geometry of the drive.) The second mode is called *LBA* addressing, which stands for Logical Block Addressing. In this mode, the sectors are numbered with a 28-bit value, starting with zero. If the harddrive is viewed as a file, the sector number multiplied by 512 would give the byte offset in the file of the beginning of that sector, for any size harddrive, and for any geometry.

2.3 Controller

For the 'brains' of a project, I chose the PIC18F452 made by Microchip. I was familiar with their PIC controllers and this particular model has a few features that are suitable for my project. It has 1.5Kbytes of data RAM (the highest of all their products). This allowed me to buffer 1-2 sectors from the harddrive without the need of external memory. I was trying to use as little hardware as possible so there would be fewer things that *would* go wrong. It also contains a synchronous serial port, which I used to send data to the VLSI 1001k MP3 decoder. Also, the PIC18F452 can run at up to 40MHz, which is equivalent to 10 million instructions per second. I initially estimated that 4-16 MHz should be sufficient.

The PIC18F452 has a slew of I/O ports — 34 of its 40 pins are general purpose I/O pins. The built-in modules (such as the Synchronous Serial Port) also use share some of the 34 I/O pins. The I/O pins are divided into ports A,B,C,D,E — each are accessed in the software as an 8 bit register. I dedicated 4 pins of Port A to function as outputs to make 3 address lines (Port A, bits 0-2) and one Write/Read- (bit 4) line. A 74LS138 3-to-8 Decoder provides enable signals for the Harddrive, LCD, MP3 Decoder, and switches. The decoder also ensures that no two devices are ever enabled at the same time. Another pin of Port A is used an input to receive the *Data Request* signal from the MP3 Decoder. Ports B and D form a 16 bit bidirectional data bus, with port B is used for the low byte and Port D for the high byte. Port E is used as an output port to provide 3 control lines. For the LCD Display, one pin of port E provides the *Register Select* line. For the harddrive, all three pins of port E are connected to the harddrive's register select lines. For the MP3 decoder, one pin of Port E is used as the BSYNC signal.

2.4 LCD Display

For displaying information about what song is playing, I chose a 2x20 character LCD display. It has two control lines, an *Enable* and a *Register Select*. The *register select* is used to tell the display whether it is getting data or an instruction. There are also 8 bidirectional data lines which are used to transmit data and instruction bytes (in parallel). There is also a *Read/Write-* line that is connected with an inverter to the *W/R-* on the PIC. The *Enable* line is also active-high and is connected through an inverter to the decoder output labeled *ADDR_LCD*. The LCD display was very helpful during the construction of the project as I was able to use it to show debugging info.

Table 1: Timing requirements for playing an mp3 file successfully

Access time of harddrive (for 1 sector)	25 ms (max) 12ms (avg)
Speed at which the harddrive can transfer data	40 sectors/sec (min)
Bytes in one sector of the harddrive	512 bytes (4096 bits)
Bit rate of mp3 files	128Kbps - 192Kbps
Sectors per second need to play an mp3	16sectors/sec - 24sectors/sec
Time it takes to play one sector (based on MP3 data-rate)	31ms (128Kbps) 21ms(192Kbps)
Buffer length in MP3 decoder	128 ms
Speed at which the PIC sends data serially	1.75 Mbits/sec (@28 MHz)
Time to send one sector to the the MP3 Decoder	3ms

2.5 Human Interface

For controlling the player, I chose a 74AHCT245 Bus Transceiver and 4 momentary switches. The 245's direction line is tied down and the *ADDR_SWITCHES* line is connected to the transceiver's *output enable* line. When the transceiver is selected, the values of the switches will placed onto the data bus. I plan to check the bus fairly often (perhaps 10 times a second), so no interrupts nor latch is necessary. The switches have the standard functions like *Play*, *Previous Song*, *Next Song*, and *Stop*. Due to time constraints, I was not able to make use of the switches during the duration of this project.

2.6 Design Timing

What follows is a general description of the hardware involved and why it was chosen. When I was in the process of deciding what hardware to use, I tried to plan out how fast everything needed to happen.

The table lists the times and data rates required to do some of the main steps in transfering an mp3 file from a harddrive to the mp3 decoder. Since the buffer length in the mp3 decoder is fairly long, no too much buffering of the harddrive is necessary. The harddrive can transfer sectors faster than the mp3 decoder needs, so buffering entire file in memory is not necessary. The access time for the harddrive listed above is the absolute worst case. The average latency is half of what is stated above, and should be even less when consecutive sectors are accessed. This means that by buffering one sector ahead, this player should be able to keep up with most high bit-rate streams. Furthermore, streams above 128Kbps are often variable bit-rate streams. Although they may have peaks of 256Kbps, they tend to have an average bit rate at or below 192Kbps. In experimental testing, I was able to sustain transfers of more than 50 sectors per second from

Table 2: Device Address Assignments

Device	Address
(Nothing)	0
Switches	1
LCD	2
HD Control Block	3
HD Command Block	4
MP3 Decoder Control Bus	5
MP3 Decoder Data Bus	6
(not necessary, but useful for debugging)	

the harddrive. From another point of view, every 21-31ms one sector must be sent to the decoder, and it takes less 3ms to send the data from the PIC to the decoder. The time required to actually move 1 sector from the drive to the PIC is fairly short, less than 1ms. This leaves a lot of time for doing other things (such as updating the LCD Display). Accessing the harddrive can be further optimized by telling the harddrive to read more than one sector at a time, but this is not necessary.

3 Procedure

Note: All software was initially implemented in assembly, and the later in C when a C compiler was obtained.

3.1 LCD Verification

The first thing I did was to connect the PIC, 3-8 decoder, and LCD together. The LCD needs an inverter since its Enable is Active High, and the 74LS138 decoder's outputs are Active-Low. It also has a Write/Read-line which needed to be connected through an inverter to the PIC. I initially used a 16MHz crystal for the PIC (4 Million instructions/sec). The data lines of the LCD were connected to the low byte data lines of the PIC (PORT B). I assigned the following addresses listed in Table 2 to the devices.

The address *Hardware Reset* means that anytime this address is chosen, all devices with reset lines will be reset – the harddrive and the MP3 Decoder. Address 0 is selected when no device is to be selected. When the *W/R-* changes or when control of the bus changes from one device to another, address 0 is selected during the transition.

I wrote low-level software routines for selecting and de-selecting a device. This included setting the appropriate address lines and the write/read- line. These routines were critical as the hardware could be damaged if they were incorrect. Then I started writing low-level LCD routines for writing a byte to the instruction and data registers (that made use of the low-level device access routines). Next I wrote the LCD initialization routine, based on the 'long' initialization specified in the data-sheets. To test the LCD display, I wrote a short program that would initialize the LCD and then display a few words on it to verify that things were working. I had a few problems with this stage because I overlooked a few critical timing parameters. Once those were fixed, the LCD worked perfectly.

3.2 Harddrive Verification

The next stage was to interface to the harddrive. This involved wiring its 16 data lines to ports B & D. Next was to connect its three address lines to Port E. Then the two enable lines were connected to the decoder

(only one enable can be asserted at a time). The harddrive has separate Write and Read lines that must be asserted AFTER it is selected. I chose to use two (for minimal extra hardware) more pins of the PIC exclusively for this purpose. The DMA Request line and the grounds were all tied to ground. All other pins were left unconnected.

The harddrive interface is very simple. It doesn't require any initialization beyond the hardware reset. Since I chose to use LBA mode, I wrote `11100000` (binary) to the *Drive/Head* register. I already knew the harddrive I used supported LBA mode by running `hdparm -I /dev/hdb` in Linux. I copied an mp3 file *[Abbott & Costello - Who's on first?]* to the absolute beginning of the harddrive by running `cp file.mp3 /dev/hdb`. This overwrote the partition table and placed the file data starting at sector 0.

I used the LCD display to display the status register of the harddrive after each command I sent to it. In this way, I was able to determine if things were going correctly. For the first test, I issued the *Identify Drive* (`0xEC`) command to the harddrive and read the data back. This would tell things like Model Name, Serial Number, etc. When the identify drive command is issued, the harddrive prepares 1 sector's worth of data in its buffer that contains information about itself. The data is read by reading the data register 256 times. I was able to successfully display the name and serial number of the harddrive on the LCD screen.

For a third test, I tried to read a sector from the harddrive. To read a sector from the harddrive, I wrote a "1" to the *Sector Count* register, and then filled the Drive/Head, *Cylinder High*, *Cylinder Low*, and *Sector Number* registers with the 28 bit LBA address of the sector I wanted to access. Then `0x20` is written to the command register which is the *Read Sector* command. The status register is read until the BSY flag in it goes low. Then the data register is read 256 times. To see that everything did work, I wrote out each byte that was read to the LCD screen in binary and compared it with the contents of the file.

For a forth test (which done much later after the MP3 decoder was wired up and after a great deal of frustration), I made a loop that consisted of reading a sector from the harddrive, incrementing a counter, and displaying the count of the counter on the LCD screen. This was to see if everything was running quickly enough. Experimentally, I was able to achieve 50 sectors/second. With these tests complete, I deemed the harddrive to be working.

3.3 MP3 Decoder Verification

I connected the mp3 decoder as the example circuit on the datasheet describes with the exception that I didn't isolate the 'analog' and 'digital' supplies and corresponding grounds. The PIC only has one Synchronous Serial Port, but the MP3 Decoder uses two. Using an example from the datasheet, I tied the two serial inputs (on the MP3 Decoder) together and gated the clock of the data bus with an AND gate. Since the 3to8 decoder is active low, by ANDing the Chip Select with the serial clock, the data bus will only receive data when the device is not selected. Strangely, the mp3 decoder must *NOT* be selected while mp3 data is being sent to it. The device is only 'selected' while a command is being sent down the control bus.

This MP3 decoder is a 3V device, not a 5V device, so I used a 74LVC245 bus transceiver to convert the 5V TTL signals for 3V operation. Its direction input is tied to ground. The 74LVC245 is a 3V device, but has 5V tolerant inputs that won't be damaged by regular 5V TTL signals. To provide 3V, I used a small 3V, 150ma regulator. This provides ample power for the bus transceiver as well as the MP3 Decoder. All inputs to the MP3 Decoder were routed through the 74LVC245, including: serial clock, serial data input (control), serial data input (data), serial data output (control), chip select, and reset. The input for *BSYNC* was connected to pin2 of Port E on the PIC. The output *DREQ* was connected to pin 4 of Port A of the PIC. The serial data output and *BSYNC* were also routed through the transceiver so that no wiring mistake could expose the MP3 Decoder to 5V.

After wiring everything up (and triple checking all the wiring), I powered up the circuit. According to the datasheet, the RCAP pin should be approximately 1.3V. I was able to verify this. This means the analog section of the MP3 decoder was functioning properly.

The datasheet for the mp3 decoder gives two possible diagnostic tests. One tests the the command bus by setting the volume control off and on. Turning the volume control all the way off makes a popping sound because it powers down part of the decoder. This is done by selecting the MP3 Decoder, sending `0x02, 0x0B, 0x00, 0x00` down the mp3 decoder's control bus, waiting approximately one second, and then sending 0x02 0x0B, 0xFF, 0xFF, and de-selecting the decoder. Repeatedly doing this step should make a continuous popping sound. When sending data to the control bus, `0x02` is the write opcode. Above, `0x0B` specifies the register to access (volume in this case), and the last two bytes are the data to write to the register. For the volume control, each byte represents the volume level for one channel (left or right).

The second test involves sending a special sequence down the data bus that causes a continuous tone to be played. (I assume this sequence is not normally found in valid mp3 streams). This is done by writing `0x53, 0xEF, 0x6E, 0x30, 0x00, 0x00, 0x00` to the data bus, waiting about a second, then writing `0x45, 0x78, 0x69, 0x74, 0x00, 0x00, 0x00` to the data bus. Repeatedly doing this (and setting *BSYNC* appropriately) should produce a beeping sound.

I initially had some problems with this test because I was using a crystal for the mp3 decoder that was slightly too fast for it. I had misinterpreted the company's claims of their product operating from 24-32 MHz to imply that it could actually operate at 28MHz. Their website lists that they only qualify their chips to 24 MHz. Buyer beware! I didn't have any 24MHz crystals on-hand, so I used 22MHz crystal. The beeping test then worked and I concluded that the mp3 decoder was working.

I also did 'speed' test similar to the one that I did on the harddrive. I made a loop that sent out 16 blocks of 32 bytes [not from the harddrive, just any data] (while respecting *DREQ*) and displayed a counter on the screen. The first time my project played any audio (from the harddrive), it was very broken up and garbled. It turned out that the C compiler didn't optimize well some of the code I wrote for the routine that sent data to the decoder. I did this test and discovered that the routines for sending data to the mp3 decoder were extremely slow. By cleaning the code up slightly, the compiler was better able to use some of the special instructions on the PIC and produce some extremely efficient code (in fact, it did exactly what I originally wrote in assembly!). The code that sends data to the decoder needs to be able to do at least 24 sectors per second. I was able to achieve 40-50 sectors per second. All that was left now was to play some audio...

3.4 Playing Audio

With these tests done, I concluded that everything was working. To play audio the following steps are done by software:

1. Initialize LCD
2. Issue hardware reset to harddrive, mp3 decoder
3. Start and finish reading first sector from harddrive.
4. Issue software reset to mp3 decoder
5. Issue command to read next sector from harddrive.
   (the harddrive will be busy reading next sector while current sector is played)
6. Wait for *DREQ* to go high.
7. Send one block of 32 bytes to the mp3 decoder (data bus).
   (This should cause sound to be heard in the headphones)
8. repeat steps 6 & 7 16 times.

9. Read data for next sector (the one described in step 5).
10. Goto step 5.

Between steps 9 and 10 is a good time to poll the mp3 decoder for info such as elapsed time, bit-rate, etc.

3.5 Conclusions

This project was a true learning experience. I spent **countless** hours in anguish because I made several difficult to find mistakes in my low-level code. I also learned the true value of a C compiler. Fortunately the good people at High-Tech software generously donated a copy of their $850 compiler to me for my project. The PIC uses a very reduced RISC instruction set. The assembler does not differentiate between immediate data and register numbers. The PIC has a very small runtime stack that cannot be manipulated. I tried to use macros to make the assembly code more readable but I also introduced many mistakes that way. Often I supplied an immediate value to a macro that was expecting a register. Of course the assembler didn't care and this was very hard to track down. As soon as I obtained the C compiler, everything worked flawlessly on the first attempt! At the For the next project I attempt of this scale, I will use a compiler from the beginning.

Some testing with an ampmeter revealed that the controller & LCD together use 20ma, the backlight of the LCD uses approx. 100ma alone, and the harddrive draws about 0.5A. The total current consumption is approximately 0.6A. This does not exclude the possibility of running this project from batteries. In the future, when CompactFlash technology improves, a CompactFlash-to-IDE adapter would allow a CompactFlash card to have an IDE interface. This would signficantly reduce current consumption.

Future improvements for this project will include making use of the switches and implementing a filesystem. I have attempted to implement the filesystem used by Windows (FAT32), while using less than 1000 bytes of RAM (in C), but it isn't very efficient. The player should be able to handle a few thousand songs without any external memory, so I will design my own filesystem. Attached with this report is my design for a filesystem.

References

[1] Linear Technology Corporation. Lt1121 micropower low dropout regulators with shutdown. 1994.

[2] Microsoft Corporation. Microsoft extensible firmware initiative: Fat32 file system specification. 2000.

[3] Optrex Corporation. Lcd module specification. 1999.

[4] Hitatchi. Hd44780u dot matrix liquid crystal display controller/driver.

[5] Microchip Technology Inc. Pic18fxx2 datasheet: High performance, enhanced flash microcontrollers with 10-bit a/d. 2001.

[6] Texas Instruments. Sn54ahct245, snahct245 octal bus transceivers with 3-state outputs. 2002.

[7] VLSI Solution Oy. Vlsi1001k - mpeg audio codec. 2002.

[8] Fairchild Semiconductor. Dm74ls138 decoder/demultiplexer. 2000.

[9] Seagate Technology. Ata interface reference manual, rev. c. 1993.

3.6 Program Listing

```
#include <pic18.h>
#include <string.h>
#include <stdio.h>


__CONFIG(1, OSCSDIS & HS );
__CONFIG(2, PWRTEN & BORV42 & BOREN & WDTDIS );
__CONFIG(4, STVREN & DEBUGDIS & LVPDIS );

#pragma printf_check(printf) const

#define XTAL_FREQ 16MHZ
#include "delay.inc"

void PrintBinary(unsigned char c);

void putchar(const unsigned char c);
static unsigned char LCD_row;
static unsigned char LCD_col;

/*=============================================================*/
/* HARDWARE INTERFACE CONSTANTS */
/*=============================================================*/
/*
    PORT A  BITS 3-0 ADDR/RW   (Bit 4 = DREQ = INPUT )
    PORT B  BITS 7-0 DATA_LOW
    PORT C  BITS 1-0 DIOR/DIOW,  5-4 SPI
    PORT D  BITS 7-0 DATA_HIGH
    PORT E  BITS  2-0 CONTROL
*/
#define DATA_LOW                PORTB
#define DATA_LOW_TRIS           TRISB
#define DATA_HIGH               PORTD
#define DATA_HIGH_TRIS          TRISD
#define PORT_ADDR               PORTA

#define PORT_CONTROL            LATE

/*=============================================================*/
/* BUS ADDRESS CONSTANTS */
/*=============================================================*/
#define ADDR_NOTHING        0
#define ADDR_SWITCHES       1
#define ADDR_LCD            2
#define ADDR_HD_CONTROL     3
#define ADDR_HD_COMMAND     4
#define ADDR_MP3_CONTROL    5
#define ADDR_MP3_DATA       6   /* Not currently Used */
#define ADDR_RESET          7


/*=============================================================*/
/*LCD Interface Constants                   */
/*=============================================================*/
static bit LCD_RS @ ((unsigned)&PORT_CONTROL*8)+0;

/*=============================================================*/
/*HDD Interface Constants                   */
/*=============================================================*/
static bit HD_NDIOW @ ((unsigned)&PORTC*8)+0;
static bit HD_NDIOR @ ((unsigned)&PORTC*8)+1;
#define HD_REGS PORT_CONTROL

static unsigned char hd_sector[512];    /* Global Data declaration */

/* COMMAND_BLOCK */
#define HD_REG_DATA             ADDR_HD_COMMAND, 0

#define HD_REG_ERROR            ADDR_HD_COMMAND, 1
#define HD_REG_FEATURE          ADDR_HD_COMMAND, 1
```

```
#define HD_REG_SECTOR_COUNT      ADDR_HD_COMMAND, 2
#define HD_REG_SECTOR_NUMBER     ADDR_HD_COMMAND, 3
#define HD_REG_LOW_CYLINDER      ADDR_HD_COMMAND, 4
#define HD_REG_HIGH_CYLINDER     ADDR_HD_COMMAND, 5
#define HD_REG_DRIVEHEAD         ADDR_HD_COMMAND, 6

#define HD_REG_STATUS            ADDR_HD_COMMAND, 7
#define HD_REG_COMMAND           ADDR_HD_COMMAND, 7

/* CONTROL_BLOCK */
#define HD_REG_ALT_STATUS        ADDR_HD_CONTROL, 6
#define HD_REG_DEVICE_CONTROL    ADDR_HD_CONTROL, 6
#define HD_REG_DRIVE_ADDRESS     ADDR_HD_CONTROL, 7


/*=============================================================*/
/*MP3 Interface Constants                                      */
/*=============================================================*/
static bit MP3_BSYNC @  ((unsigned)&PORT_CONTROL*8)+2;
static bit MP3_DREQ  @  ((unsigned)&PORTA*8)+4;
#define MP3_REG_MODE            0
#define MP3_REG_STATUS          1
#define MP3_REG_CLOCKF          3
#define MP3_REG_DECODETIME      4
#define MP3_REG_AUDATA          5
#define MP3_REG_HDAT0           8
#define MP3_REG_HDAT1           9
#define MP3_REG_VOL             11



/*=============================================================*/
/* Extremely Low Level General I/O Functions (Layer 1)      */
/*=============================================================*/
void SelectWriteAddress(const unsigned char addr);
void SelectWriteNothing(void);
void SelectReadAddress(const unsigned char addr);
void SelectReadNothing(void);
void Delay100ns(void);
unsigned char LCD_GetCursorPos(void);

void SelectWriteAddress(const unsigned char addr)
{
        SelectWriteNothing();
        PORT_ADDR = (addr & 7) | 8; // Bit 4 is the W/R- bit
}

void SelectReadAddress(const unsigned char addr)
{
        SelectReadNothing();
        PORT_ADDR = (addr & 7);  // Bit 4 is R- bit!
}


void SelectReadNothing()
{
        PORT_ADDR = 0;
        DATA_HIGH_TRIS = 0xFF;    // INPUTS
        DATA_LOW_TRIS  = 0xFF;    // INPUTS
}

void SelectWriteNothing()
{
        PORT_ADDR = 8;            // Bit 4 is W/R- Bit!
        DATA_HIGH_TRIS = 0x00;    // OUTPUTS
        DATA_LOW_TRIS  = 0x00;    // OUTPUTS
}


/* Even for a 40mhz PIC, this will be at least 100 ns */
```

```
// For a 28 Mhz Pic, this will be 142ns
#define Delay100ns() asm("NOP");


/*=============================================================*/
/* Low level LCD I/O Functions (Layer 2)                       */
/*=============================================================*/
void LCD_WaitBusy(void);
void LCD_PutChar(const unsigned char c);
void LCD_PutInstr(const unsigned char c);
void LCD_SetCursor(const unsigned char pos);
void LCD_Clear(void);
void LCD_RowCol(const unsigned char row, const unsigned char col);
void LCD_Init(void);

void LCD_WaitBusy()
{
        static unsigned char x;
        LCD_RS = 0; /* Instruction */
        do
        {
                SelectReadAddress( ADDR_LCD );
                Delay100ns(); /* Delay at least 40 ns */
                x = DATA_LOW;
                SelectReadNothing();
        } while (  x & 128 ); /* While busy */
}

unsigned char LCD_GetCursorPos()
{
        static unsigned char x;

        LCD_RS = 0; /* Instruction */

        LCD_WaitBusy();

        SelectReadAddress( ADDR_LCD );
        Delay100ns(); /* Delay at least 40 ns */
        x = DATA_LOW;
        SelectReadNothing();

        return (x & 127);
}



void LCD_PutChar(const unsigned char c)
{
        LCD_WaitBusy();
        SelectWriteNothing();
        DATA_LOW = c; /* Put data on outputs */

        LCD_RS = 1; /* Data */
        SelectWriteAddress( ADDR_LCD );
        Delay100ns(); /* Delay at least 40 ns */
        SelectWriteNothing();

        Delay100ns();
        Delay100ns();
        Delay100ns();
        Delay100ns(); /* Min Delay of 230 ns */
        LCD_col++;
}


/* Not not wait for LCD to become Un-Busy */
void LCD_PutInstr(const unsigned char c)
{
        SelectWriteNothing();
        DATA_LOW = c; /* Put instr on output */
```

```
        LCD_RS = 0;   /* 0 = instr */
        SelectWriteAddress( ADDR_LCD );
        Delay100ns(); /* Delay at least 40 ns */
        SelectWriteNothing();

        Delay100ns();
        Delay100ns();
        Delay100ns(); /* Min Delay of 230 ns */
}


void LCD_SetCursor(const unsigned char pos)
{
        LCD_WaitBusy();
        LCD_PutInstr(  128 + ( pos & 127 )  );
}


void LCD_Clear()
{
        LCD_WaitBusy();
        LCD_PutInstr(1);
        DelayMs(2);  /* 1.6 ms Delay */
        LCD_row=0;
        LCD_col=0;

}


void putchar(const unsigned char c)
{

        if (c=='\n')
        {
                LCD_row++;
                LCD_col=0;
        }
        else
        {
                if ( LCD_col >= 20 )
                {
                        LCD_col=0;
                        LCD_row++;
                }


                if ( LCD_row >=2 )
                {
                        LCD_row=0;
                        LCD_col=0;
                        LCD_Clear();
                }

                LCD_SetCursor( LCD_row*0x40 + LCD_col );
                LCD_PutChar(c);
        }

}

/* Public function for moving cursor */
void LCD_RowCol(const unsigned char row, const unsigned char col)
{
        LCD_row=row;
        LCD_col=col;
        LCD_SetCursor( row*0x40 + col );
}


void LCD_Init()
{
        LCD_PutInstr(0b00110000);  DelayMs(5);          /* Function Set 1/3            */
        LCD_PutInstr(0b00110000);  DelayUs(150);        /* Function Set 2/3            */
```

```
        LCD_PutInstr(0b00110000);  DelayUs(150);        /* Function Set 3/3             */
        LCD_PutInstr(0b00111100);  DelayUs(50);         /* Function Set: 8 bit, 2 lines */
        LCD_PutInstr(0b00001000);  DelayUs(50);         /* Display Off                  */
        LCD_PutInstr(0b00000001);  DelayMs( 3);         /* Display CLEAR                */
        LCD_PutInstr(0b00000110);  DelayUs(50);         /* Entry Mode Set               */
        LCD_PutInstr(0b00001111);  DelayMs( 5);         /* Display On                   */

}

/* ===================================================================== */
/* HDD LOW - LEVEL Routines                                                                                                  */
/* ===================================================================== */

unsigned char  HD_ReadByte(const unsigned char addr, const unsigned char reg);
void HD_WriteByte(const unsigned char addr, const unsigned char reg, const unsigned char dat);
void HD_Read512(void);
void HD_WaitBusy(void);



unsigned char HD_ReadByte(const unsigned char addr, const unsigned char reg)
{
        static unsigned char d;

        SelectReadNothing();
        HD_NDIOR = 1;
        HD_NDIOW = 1;

        HD_REGS = reg;

        SelectReadAddress( addr  );
        Delay100ns();
        HD_NDIOR = 0; /* Turn ON */
        Delay100ns();
        Delay100ns();
        d = DATA_LOW;
        HD_NDIOR = 1; /* Turn Off */
        Delay100ns();
        SelectReadNothing();
        Delay100ns();
        return d;
}


void HD_WriteByte(const unsigned char addr, const unsigned char reg, const unsigned char dat)
{
        HD_WaitBusy();
        SelectWriteNothing();
        HD_NDIOR = 1;
        HD_NDIOW = 1;

        HD_REGS = reg;
        DATA_LOW = dat;

        SelectWriteAddress(addr);
        Delay100ns();
        HD_NDIOW = 0; /* Turn On */
        Delay100ns();
        HD_NDIOW = 1; /* Turn Off*/
        Delay100ns();
        SelectWriteNothing();
}


/* Assumes address of harddrive is ADDR_HD_COMMAND */
/* Assumes data is 'hd_sector' (512) */
void HD_Read512()
{
        static unsigned short i;

        HD_WaitBusy();
```

```
        HD_NDIOR = 1;
        HD_NDIOW = 1;
        HD_REGS = 0; /* Data */


        SelectReadAddress( ADDR_HD_COMMAND );
        Delay100ns();
        Delay100ns();

        for (i=0; i<512;)
        {
                Delay100ns();
                HD_NDIOR = 0; /* Turn On */
                Delay100ns();
                Delay100ns();
                Delay100ns();
                hd_sector[i++] = DATA_LOW;
                hd_sector[i++] = DATA_HIGH;
                HD_NDIOR = 1; /* Turn Off */
                Delay100ns();
        }
        Delay100ns();
        SelectReadNothing();
}


void HD_WaitBusy()
{
        static unsigned char x;
        do
        {
                x=HD_ReadByte( HD_REG_STATUS );
        }
        while (x & 128);
}


void HD_StartReadSector(const unsigned long sector)
{
        static unsigned char s0,s1,s2,s3;

        s0=sector % 256;
        s1=(sector>> 8) % 256;
        s2=(sector>>16) % 256;
        s3=(sector>>24) % 256;
        s3= 0b11100000 | (s3 & 0b00001111);     // LBA, Drive0
        HD_WriteByte( HD_REG_SECTOR_NUMBER, s0);
        HD_WriteByte( HD_REG_LOW_CYLINDER , s1);
        HD_WriteByte( HD_REG_HIGH_CYLINDER, s2);
        HD_WriteByte( HD_REG_DRIVEHEAD    , s3);
        HD_WriteByte( HD_REG_COMMAND,      0x20); // Read Sector
}

/* ************************************************************************** */
/* ************************************************************************** */
/* SPI                                                                        */
/* ************************************************************************** */
unsigned char SPI_SendByte(const unsigned char c);
void SCI_Write(const unsigned char reg, const unsigned short value);
short SCI_Read(const unsigned char reg);
void SDI_Write(const unsigned dat);

unsigned char SPI_SendByte(const unsigned char c)
{
        static unsigned char r;
        MP3_BSYNC=0;
        while ( !(SSPSTAT & 1) ); // Wait for data to be transmitted
        r=SSPBUF;
        MP3_BSYNC=1;
        SSPBUF = c;
        asm("NOP");asm("NOP");  asm("NOP"); asm("NOP"); asm("NOP");
        MP3_BSYNC=0;
```

```
        return r;
}


void SCI_Write(const unsigned char reg, const unsigned short value)
{
        static unsigned char hiByte, lowByte;
        hiByte = value/256;
        lowByte= value%256;

        SelectWriteAddress( ADDR_MP3_CONTROL );

        SPI_SendByte( 0b00000010 ); // Write Data
        SPI_SendByte( reg );
        SPI_SendByte( hiByte );
        SPI_SendByte( lowByte);
        SelectWriteNothing();
        DelayUs(5);
}

short SCI_Read(const unsigned char reg)
{
        static unsigned short value;
        static unsigned char hiByte, lowByte;

        SelectReadAddress( ADDR_MP3_CONTROL );

        SPI_SendByte( 0b00000011 ); // Read Data
        SPI_SendByte( reg );

        SPI_SendByte(0);
        hiByte =SPI_SendByte(0);
        lowByte=SPI_SendByte(0);

        SelectReadNothing();
        value = hiByte*256 + lowByte;
        return value;
}

void SDI_Write(const unsigned dat)
{
        SelectWriteAddress( ADDR_MP3_DATA );
        SPI_SendByte( dat );
        SelectWriteNothing();
}

void MP3_SoftReset(void)
{

        SCI_Write( MP3_REG_CLOCKF, 11059 ); //22mhz
        SCI_Write( MP3_REG_MODE, 0b0000000000000100);
        while( !MP3_DREQ );

        SDI_Write( 0 );
        SDI_Write( 0 );
        SDI_Write( 0 );
        SDI_Write( 0 );
}



/* ************************************************************************** */
/* ************************************************************************** */
/* ************************************************************************** */
/* ************************************************************************** */


void PrintBinary(unsigned char c)
{
        static unsigned char i;
```

```
        for (i=0; i<8; i++)
        {
                if (c & 128)
                {
                        putchar('1');
                }
                else
                {
                        putchar('0');
                }

                c = c << 1;
        }
}

void main()
{

/* Disable EVERYTHING */
        INTCON=0;

        DelayUs(25);


        CCP1CON=0;
        CCP2CON=0;
        SSPCON1=0;
        SSPCON2=0;
        TXSTA=0;
        RCSTA=0;
        ADCON0=0;
        ADCON1=0x05;
        T0CON=0;
        T1CON=0;
        T2CON=0;
        T3CON=0;
        LVDCON=0;


// INITIALIZE TRIS registers.....
        TRISA = 0b11110000; /* address port is all outputs, provide some inputs*/
        TRISB = 0xFF;           /* Low  Data is all INPUTS                      */
        TRISD = 0xFF;           /* High Data is all INPUTS                      */
        TRISE = 0x00;           /* control lines are all outputs                */
        PORTC = 0xFF;           /* All High */
        TRISC = 0B11010100;     /* DIOR/DIOW are outputs, SPI done correctly    */

        SelectReadNothing();

// Initialize SPI port...


//               76543210
        SSPCON1=0b00100001;     // Master Mode, Enable SPI, idle state is LOW, 1/16 fsoc
        SSPSTAT=0b01000000;     // Master,Xmit on Rising Edge, Sample in middle,
        SSPBUF =0 ;             // transmit dummy


/* Issue General Reset */
        DATA_LOW=0;
        DATA_HIGH=0;
        SelectWriteAddress( ADDR_RESET );
        DelayMs(100);
        SelectWriteNothing();
        DelayMs(100);

/* Initialize LCD */
        LCD_Init();

/* Initialize HD  */
// Set LBA, Drive 0
                HD_WriteByte( HD_REG_DRIVEHEAD, 0b11100000 ); // LBA, drive0
```

```
// Set Sector Count
                HD_WriteByte( HD_REG_SECTOR_COUNT, 1);

// Play
        {
                unsigned long secNum=0;
                unsigned char i, j;
                unsigned short index;
                unsigned short decodetime;

                secNum=0;
                HD_StartReadSector(secNum);
                HD_Read512();

                MP3_SoftReset();

                while(1)
                {
                        secNum++;
                        HD_StartReadSector(secNum);

                        index=0;
                        for (i=0; i<16; i++) // Write Data to mp3 Decoder...
                        {
                                while (MP3_DREQ == 0); // While not ready to receive...

                                for (j=0; j<32; j++)    // Send a 32 byte block
                                {
                                        SDI_Write( hd_sector[index] );
                                        ++index;
                                }
                        }// for i...

                        decodetime = SCI_Read( MP3_REG_DECODETIME);
                        LCD_RowCol(0,0);
                        printf("Elapsed %2d:%2d", decodetime/60, decodetime%60);

                        HD_Read512();   // Finish reading the next sector

                } // end while(1)
        }

        asm("sleep");
}
```