

SOLID-STATE MP3 PLAYER

By

MOHD FAIZUL FATAN ABDUL RAHMAN

FINAL PROJECT REPORT

Submitted to the Electrical & Electronics Engineering Programme
in Partial Fulfillment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

© Copyright 2005

by

Mohd Faizul Fatan Abdul Rahman, 2005

CERTIFICATION OF APPROVAL

SOLID-STATE MP3 PLAYER

by

Mohd Faizul Fatan Abdul Rahman

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved:




Mr. Patrick Sebastian
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

December 2005

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



Mohd Faizul Fatan Abdul Rahman

ABSTRACT

The topic for the Final Year Research Project (FYP) is “Solid-State MP3 Player”. A solid-state memory is introduced here to replace the existing compact disc that is used to store data. The project requires a microcontroller-based interface circuit to control the player.

A simple block diagram of a complete player basically consists of

- A solid-state memory to store data.
- Initialize the storage into the mode of transferring the files.
- The microcontroller that control the data transfer between storage and microcontroller.
 - i. Data transfer to storage.
 - ii. Data transfer from storage.
- A decoder to run a decompression algorithm that undoes the compression of the MP3 file and then a digital-to-analog converter turns the bytes back into waves.
- An amplifier to boost the strength of the signal and sends it to the audio port, where a pair of speakers is connected.

The project mainly covers the manipulation of programming language to implement the routines in the microcontroller.

ACKNOWLEDGEMENTS

First and foremost, all praise to Allah s.w.t for granting me the opportunity to complete this final year project, which has proven to be a very enriching experience.

It is with pleasure that I express my heartfelt thanks to all who have assisted me either directly or indirectly during the course of this project. My gratitude goes to my supervisor, Mr. Patrick Sebastian, who helped me achieve my project objectives. I would like to acknowledge that without his guidance, all my efforts would not have been fruitful.

I would also like to thank my partner, Ms. Harni Farihah Mohd Safari Lai, who has been working with me through thick and thin in making this project a success. To Miss Nasreen, the FYP coordinator, thank you for your help in explaining all the proper procedures for this project.

Finally, I forward my special thanks to Mr. Zuki, the lecturer of Microcontroller, my friends and family for their unwavering support during this project. Not to forget anyone whose name I did not mention here. Your contribution is greatly valued.

TABLE OF CONTENTS

LIST OF TABLES.....	ix
LIST OF FIGURES	x
LIST OF ABBREVIATIONS.....	xi
CHAPTER 1 INTRODUCTION.....	1
1.1 Problem Statement.....	1
1.2 Objectives and Scope of Study	1
CHAPTER 2 LITERATURE REVIEW AND THEORY.....	3
2.1 MP3 over CD	3
2.2 MP3 Player.....	4
2.2.1 Microcontroller.....	5
2.2.2 STA013 MPEG Decoder.....	6
2.2.3 CS4334 DAC.....	6
2.2.4 MultiMedia Card	7
CHAPTER 3 METHODOLGY	8
3.1 Project Methodology.....	8
3.2 Hardware Design	9
3.2.1 System Communication	9
3.2.2 Printed Circuit Board Design	10
3.3 PICC Software Design.....	10
3.3.1 Subsystem Libraries	11
3.3.2 Main PIC Code.....	11
3.3.3 STA013 Library	12
3.3.4 MMC Library	12
3.4 WARP 13 Board	12
CHAPTER 4 RESULTS & DISCUSSION	14
4.1 Setup PIC16F877	15
4.2 LEDs	15
4.3 SPI Interface Overview.....	15
4.4 SPI Interface Test Module	16
4.4.1 Basic test without MMC card.....	16
4.4.2 Initialize the MMC	17

4.4.3 Connecting the MMC to the SPI	17
4.4.4 File transfer test with the present of MMC card	19
4.4.5 Intel-Hex File	19
4.5 Send data	20
4.5.1 PIC to MMC	20
4.6 Output	22
4.7 Current circuit connection.....	22
CHAPTER 5 CONCLUSION	23
CHAPTER 6 RECOMMENDATION.....	24
REFERENCES	25
APPENDICES	26
Appendix A Codes for spi hardware test	27
Appendix B GLOBAL.H	28
Appendix C MMC2.H	29
Appendix D MMC.C	31
Appendix E STA013.H	33
Appendix F STA013.C	35
Appendix G PIC.H.....	39
Appendix H PIC.C	40

LIST OF TABLES

Table 1 System Components	4
Table 2 Subsystem Communication	10
Table 3 Subsystem and Associated Software Libraries.....	11
Table 4 Pin function between PIC and MMC	18

LIST OF FIGURES

Figure 1 How the MP3 Cycle Works.....	3
Figure 2 System Block Diagram.....	4
Figure 3 PIC16F877 Pin Assignment.....	5
Figure 4 STA013 MPEG Decoder Chip.....	6
Figure 5 Project Activities Flowchart.....	8
Figure 6 STA013 Chip and Aries Adaptor Board	10
Figure 7 WARP 13 Board.....	13
Figure 8 Program Flowchart.....	14
Figure 9 SPI Communication Scheme.....	16
Figure 10 SPI Hardware Test.....	17
Figure 11 Interfacing 5 volt output to 3volt tolerance input.....	18
Figure 12 SPI Master (PIC) and Slave (MMC) communication.	19
Figure 13 Single line of Intel-Hex file.....	20
Figure 14 Tell MMC to go to SPI modes	21
Figure 15 Output to Port D	21
Figure 16 Current Circuit Connection	22

LIST OF ABBREVIATIONS

MP3 – MPEG Layer III

MMC – MultiMedia Card: Solid-state removable memory used for storage of audio tracks

LCD – Liquid Crystal Display

CD - Compact Disc

CDRW – Compact Disc Rewritable

IC – Inter Integrated Circuit

SPI – Synchronous Peripheral Interface

PIC – Microchip PIC Microcontroller (PIC16F877)

CHAPTER 1

INTRODUCTION

The idea of utilizing MP3 files away from the computer is one that has existed in the minds of many from the beginning of the introduction of MP3's. With the current technology available in the market, MP3 files can now be played with the aid of portable MP3 players. Many people have switched to MP3 players due to its mobility and new available technologies. In general, the project will deal with the design and development of an MP3 player. Enhancements are made to the existing MP3 player to allow it to read data from other sources, instead of compact-disc (CD). This project will go through a complete procedure that involves programming and hardware implementation.

1.1 Problem Statement

The purpose of having this project is due to current situation where MP3 files are stored on compact discs (CD) or rewriteable CD (CDRW). The CD media is sensitive to excessive heat and physical scratches that affect data integrity. The application of CDs also give rise to moving parts such as reading heads that would probably skip tracks when jostled.

1.2 Objectives and Scope of Study

The objectives of this project are:

- i) To study the basic principles underlying the operation of MP3 player components such as storage element, microcontroller, decoder and output.
- ii) To design and implement a portable solid-state MP3 player. The product utilizes removable storage in the form of a MultiMedia Card for storing MP3 files.

- iii) To develop codes for initialization of decoder, MMC, and to allow communication between MMC and decoder through the microcontroller.

The scope of studies has been restricted to the technology of a normal MP3 player that uses MMC as its data storage.

CHAPTER 2

LITERATURE REVIEW AND THEORY

2.1 MP3 over CD

MP3 comes from MPEG Audio Layer-3, an MPEG compression system that includes a subsystem to compress sound. MP3 can compress a song by a factor of 10 or 12 and still retain something close to CD quality. In normal CD, music is stored using 44,100 samples per second, 16 bits per sample and two channels (for stereo sound). This means that a CD stores about 10 megabytes of data per minute of music on the CD. A three-minute song therefore requires 30 megabytes of data. MP3 can compress a song by a factor of 10 or 12 and still retain something close to CD quality. So a 30-megabyte sound file from a CD reduces to 3 megabytes or so in MP3.

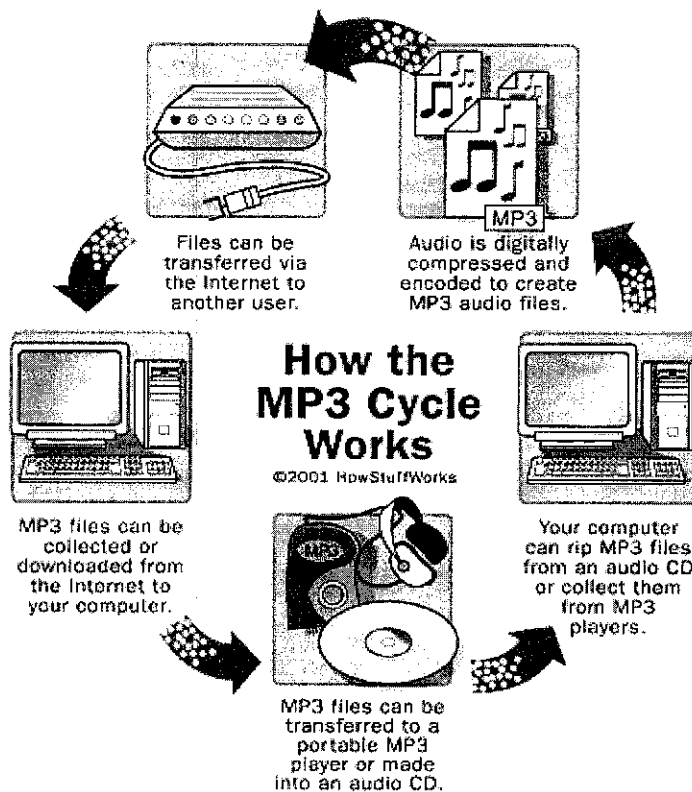


Figure 1 How the MP3 Cycle Works

2.2 MP3 Player

In this project, there are four main parts that have been identified to play major roles in ensuring a stand alone car MP3 player works. Table 1 shows the actual components used for the major systems.

System	Part	Purpose
Microcontroller	PIC16F877	Controls entire system
MPEG Decoder	STA013	Decodes MP3 bit stream
DAC	CS4334	Renders audio into analog format
MMC	MultiMedia card	Stores MP3 bit streams

Table 1 System Components

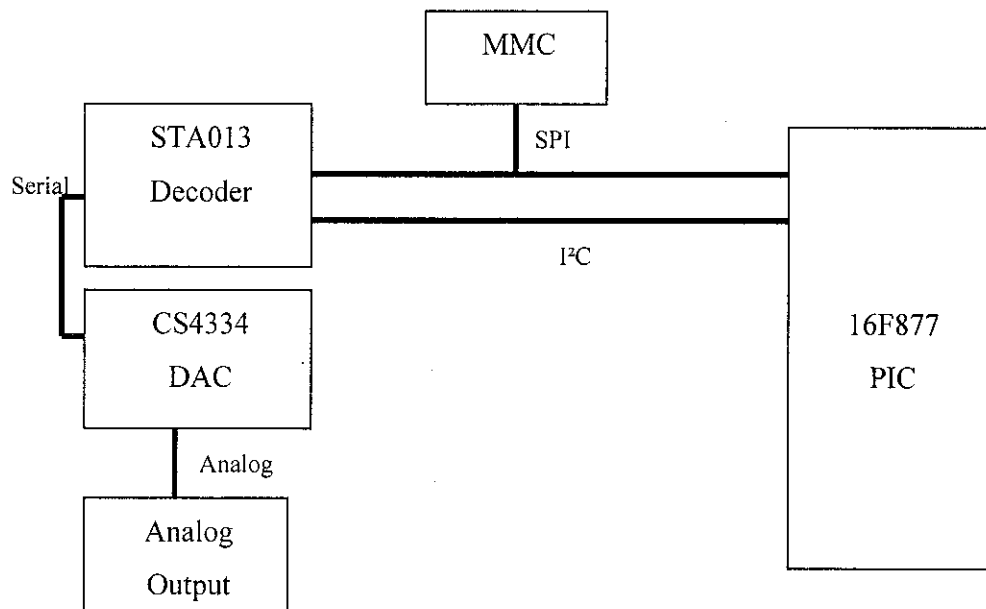


Figure 2 System Block Diagram

2.2.1 Microcontroller

There are a few numbers of microcontrollers that can be used in this project i.e. ATMEL 8051, ATMEL AVR, PIC18 and PIC16. However, due to the availability of the chips and burner, PIC16F877 has been chosen as it is available in the lab. The PIC will interface directly to all other subsystems and manage the data flow for the system as a whole. Communication with the decoder is done via the I²C interface. PIC will also interface to the MMC to stream data to the decoder using the SPI protocol. Normal operation will be for the PIC to stream data from the MMC to the MPEG decoder.

The PIC16F877 has 5 digital I/O ports (A – E) each between 3 and 8 bits wide. Each port is mapped into the register file space, and may be read/written to like any other register. The circuitry is such that it is not possible to physically input to and output from a particular pin simultaneously. For most ports, the I/O pins direction (input or output) is controlled by the data direction register, called the TRIS register. TRIS <x> controls the direction of PORT <x>. A '1' in the TRIS bit corresponds to pin acting as input, while a '0' corresponds to pin being an output.

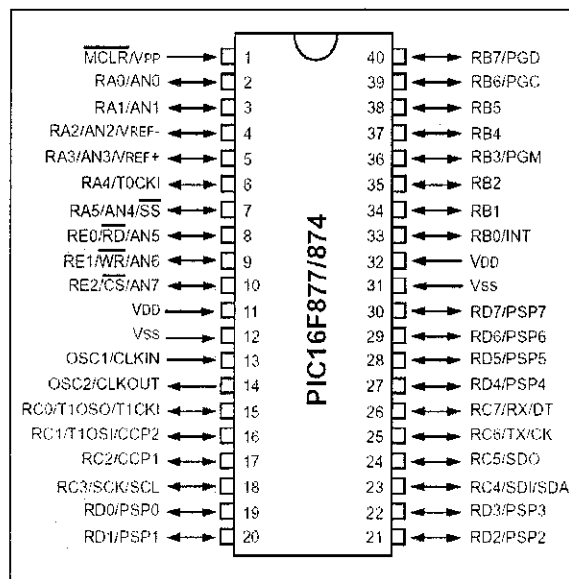


Figure 3 PIC16F877 Pin Assignment

The PORT register is the latch for the data to be output. When the PORT is read, the device reads the levels present on the I/O pins.

2.2.2 *STA013 MPEG Decoder*

The STMicroelectronics STA013 MPEG decoder is used to decode the MPEG bit stream into a format that can be rendered by a DAC. MPEG data is received via the SPI interface and the device is controlled via the I²C interface. Both of these two interfaces are relatively simple and fully supported by the PIC microcontroller. The decoded bit stream is output in serial format compatible with most commercial DACs. In total, there are only 28 pins on the device, making it small enough for use on a printed circuit board. The STA013 supports all MP3 sampling frequencies and bit rates making it ideal for this project.

The major drawback for this component is that it requires a specific configuration file to initialize it. With the limited space in the microcontroller, this configuration file has to be stored inside the MMC and later is called by the microcontroller to complete the initialization process.

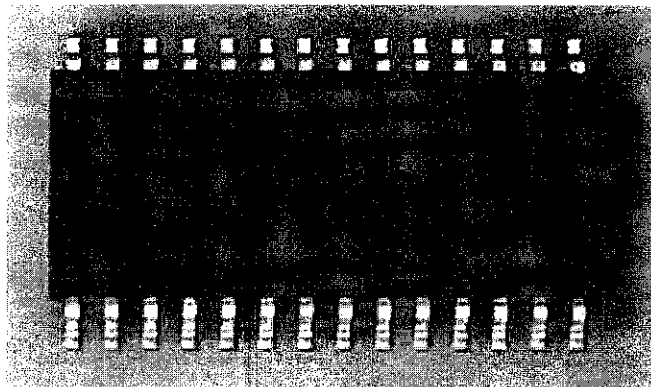


Figure 4 STA013 MPEG Decoder Chip

2.2.3 *CS4334 DAC*

The Crystal CS4334 stereo DAC is used to render the decoded bit stream into an analog audio signal. Data, supplied by the STA013, is received in serial

format and the analog representation is output. The DAC supports data precision up to 24-bits allowing it to work with the decoder, which can be configured for 16-bit, 18-bit or 24-bit. As with the decoder, the chip is available in single quantities making it an attractive choice over many other DACs.

This device is that it requires a 5.0 V supply voltage to run. This differs as the majority of the system is 3.3 V so this required the support of both voltages. Logic levels however, can be transferred between 3.3 V and 5.0 V devices with no level shifting required.

2.2.4 *MultiMedia Card*

The SanDisk MultiMedia Card is used to store the songs to be played on the MP3 player. The MultiMedia Card is access by the PIC with an SPI interface. The MultiMedia Card is initially in Multimedia mode, but when initializing it, it is set into SPI mode. The MultiMedia card can have data read from any address as long as the read data address does not cross a sector boundary. Also, when writes are made to the MultiMedia Card the write address must start at a sector boundary.

CHAPTER 3 METHODOLOGY

3.1 Project Methodology

The project involves three major objectives, namely the study of a power system model, the code writing and the construction of hardware. The flow of activities is illustrated in the flowchart.

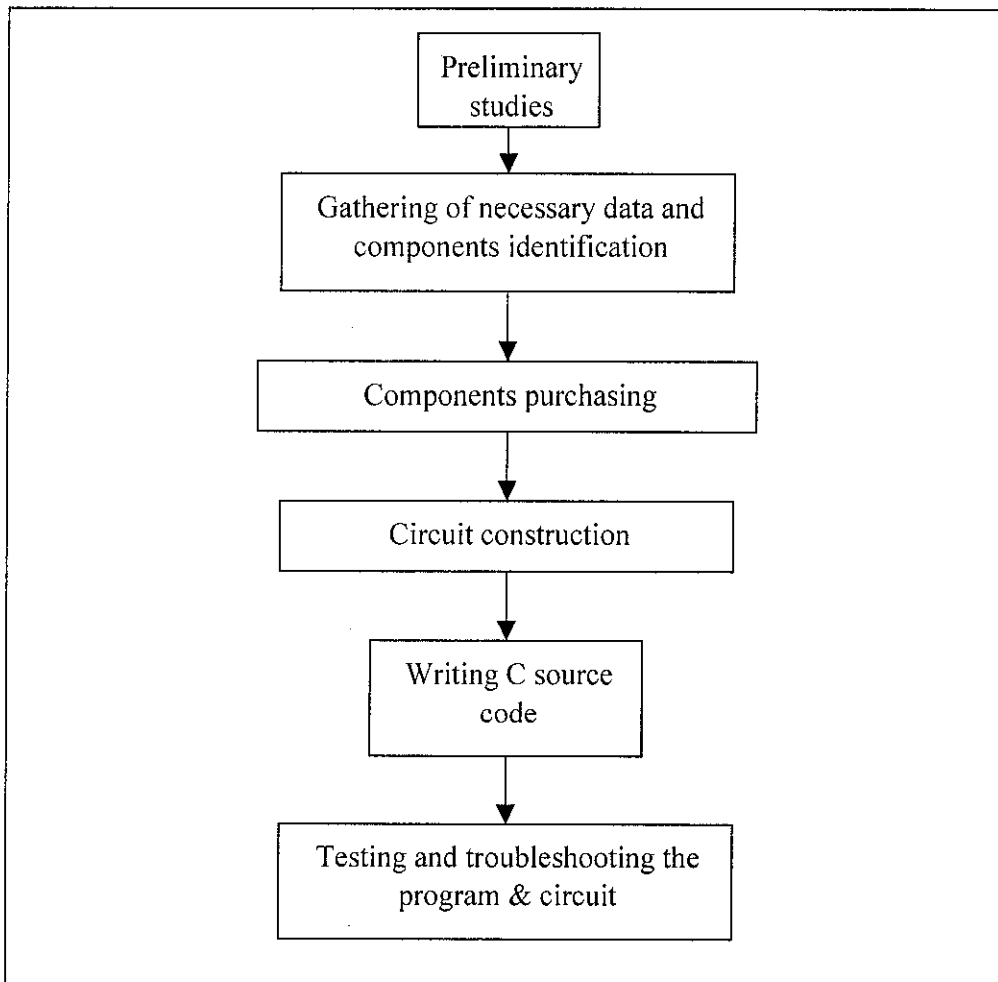


Figure 5 Project Activities Flowchart

The project has gone through a few phases of procedures. The main activity conducted during the first 8 weeks of the project was literature review concerning the technology of MP3 player. This was to gain the necessary knowledge of the components involved before embarking on the design and enhancement phase of the project. The literature review was done by studying various projects and texts from the university resource center and also from the internet.

The required components are identified and assembled in constructing the circuit. However, the STA013 and CS4334 need to be purchased over the internet (www.pjrc.com) as none of the chips are available in the market.

3.2 Hardware Design

The parts used in this project are:

- PIC16F877 Microcontroller
- ST MicroElectronics STA013 (MPEG decoder)
- Crystal/Cirrus Logic CS4334 (18 bit serial DAC)
- SanDisk Multimedia Card (16MB serial flash card)
- 14.85 MHz crystal for decoder
- 4 MHz oscillator for MCU
- Resistors, capacitors, bipolar transistors, diodes, and connectors

3.2.1 System Communication

Each subsystem requires a specific form of communication to allow it to communicate with other subsystem in this project. These communication protocols are listed below, together with the respective associated subsystems.

System 1	Communication	System 2
PIC	Serial - SPI	MMC
MMC	Serial - SPI	STA013 Decoder
PIC	Serial - I ² C	STA013 Decoder
STA013 Decoder	Serial	DAC
DAC	Analog	Output Speaker

Table 2 Subsystem Communication

3.2.2 Printed Circuit Board Design

The first challenge with using the STA013 for a prototype MP3 player is that it is a surface mount part. The two basic approaches are to buy or make an adaptor board, or to solder wires to each pin. Solder wires to each pin is not a good approach as it is not tedious and the tendency of the legs to be broken are high due to the pin sensitivity. To overcome this, a simple PCB has been designed for the STA013 and CS4334.

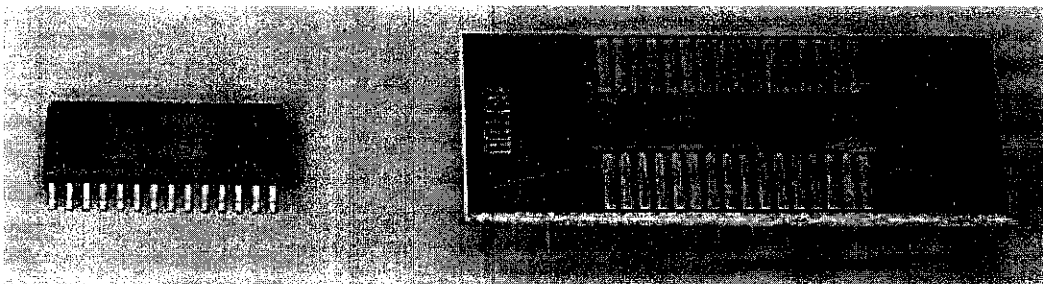


Figure 6 STA013 Chip and Aries Adaptor Board

3.3 PICC Software Design

For programming the microcontroller, the code is written in C language and compiled using the CCS PICC Compiler. PICC is a Windows-based PC application that provides a platform for developing C language code for Microchip's PICmicro microcontroller (MCU) families. Generically, PICC will program the microcontroller routine by using C language instead of assembly. The compiling process will convert

the C language into hex files that is used to download all the routines into the microcontroller. This software requires dedicated software to download the data into the microcontroller, which is the WARP13.

There are two ways to write the code, either in C language or assembly language. Programming using C language is preferred. The main reason is that the initialization of STA013 requires a big size of routines and it is very complicated and time consuming to write in assembly language. Another reason is that the PICC compiler offers several built-in functions that can be used directly to help speed programming.

To work with the software, it is preferable that the programmer has basic knowledge in C or C++ language. The datasheet of the selected microcontroller series is also needed as several important parameters can be referred to it.

3.3.1 *Subsystem Libraries*

Table below list the subsystem of this project and its respective library. Each library has its own function and they are frequently interacted with each other. A complete listing of the library interfaces can be found in the appendixes.

Subsystem	Software Library	Purpose
PIC	PIC	Main system code
MPEG Decoder	STA013	Initialization and management of STA013 MPEG Decoder
MMC	MMC	Initialization and data access of MMC

Table 3 Subsystem and Associated Software Libraries

3.3.2 *Main PIC Code*

The main PIC code contains all initialization of subsystems. It is responsible for running the entire system. Since most of the operations are handled directly by the respective libraries, this main PIC code is basically used for checking error conditions

and invoking the proper routines.

3.3.3 *STA013 Library*

This library is mainly used for the operation of the STA013 MPEG decoder. It contains the initialization of the decoder as well as the routines to control the chip. All communication to the decoder is performed over the I²C interface. In initializing the decoder, 2007 values are written into internal registers on the chip. Due to memory limitations of the PIC, these values are stored on the MMC and called on demand by the STA013 library's initialization routine. Additional function available in the library is the ability to start and stop the decoder on user's demand.

3.3.4 *MMC Library*

The MMC uses an SPI interface to control the card and transfer data. It is a four-line interface: SS, SCLK, SDI, and SDO. To send a command to the card, the user must lower SS, put data on the SDI line, and toggle SCLK (data is sampled on the rising edge for the MMC). After receiving a command, the MMC will respond with a response byte, followed by any data that was requested. A specific initialization sequence must be followed on power-up to the card in order to put the MMC in SPI mode.

3.4 WARP 13 Board

The WARP-13 is a device programmer designed to program the PIC microcontrollers. The WARP-13 is a combination of very refined firmware and software designed to provide high levels of utility function, programming speed, reliability and ease of use. This programmer requires an external power supply of 12VAC or 16VDC. It has a diode bridge on the input, so it accepts both AC and DC.

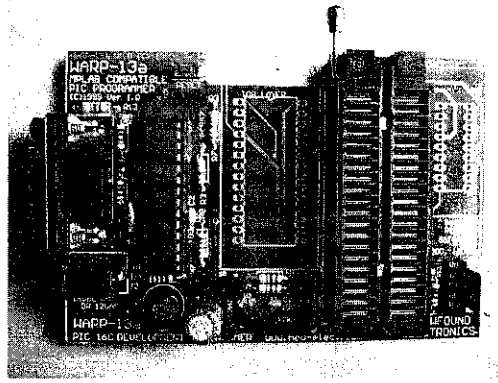


Figure 7 WARP 13 Board

CHAPTER 4

RESULTS & DISCUSSION

In accomplishing this project, the tasks are separated into two:

- interface the MMC and PIC using SPI
- interface the STA013 and PIC using I²C

This section will focus on the process of interfacing the PIC and MMC using SPI.

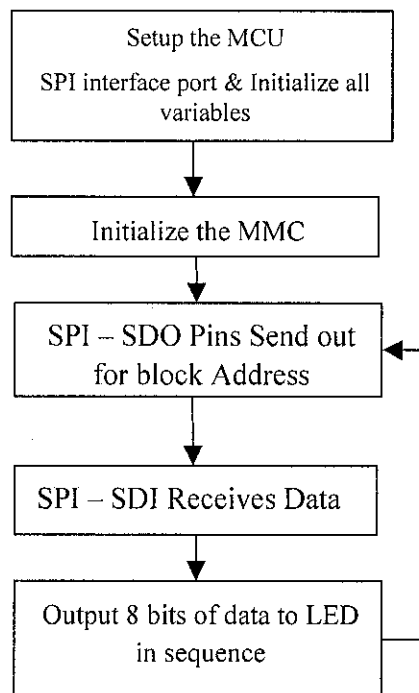


Figure 8 Program Flowchart

4.1 Setup PIC16F877

A complete coding for PIC can be found in the Appendix section. The codes are written to initialize all the variables, define ports, tris and clock. A simple program is created to control flow of data, initialize the MMC and stream the data out to the LED.

4.2 LEDs

A simple LEDs code has been added to the main PIC code and a LED is connected to Port D of PIC. These LEDs will blink continuously to indicate that the data transfer is working. The purpose of introducing this LEDs function is to help in troubleshooting the circuit. Troubleshooting the circuit takes a lot of time as there are a lot of things to be considered when the expected output cannot be seen.

4.3 SPI Interface Overview

Synchronous Peripheral Interface (SPI) is a serial communication bus, meaning that the transmitter and receiver involved in this protocol must use the same clock to synchronize the detection of the bits at the receiver.

SPI communication involves a master and a slave. Both the master and a slave send and receive data simultaneously, but the master is responsible to provide the clock signal for the data transfer. In this way, the master has control of the speed of data transfer.

Figure 9 shows the connections between the master and the slave units for SPI communication. The master supplies the clock on the SCLK pin and 8 bits of data, which are shifted out of the *serial data out* (SDO) pin. The same 8 bits are shifted into the slave (1 bit per clock pulse) on its *serial data in* (SDI) line. As the 8 bits are shifted out to the master and into the slave, 8 bits also are shifted out of the slave on its SDO line and into the master on its SDI pin. SPI communication, then, is essentially a circle in which 8 bits flow from the master to the slave and a different 8 bits flow from the slave to the master. In this way, the master and a slave can exchange data in a single communication.

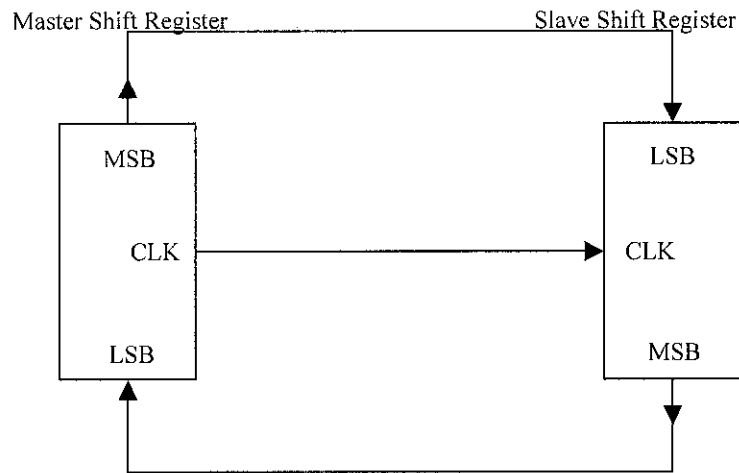


Figure 9 SPI Communication Scheme

4.4 SPI Interface Test Module

This section is important to make sure fully understanding on how SPI interfaced the MMC card to send data to the decoder. Serial data transfer is important characteristic of SPI which requires only two pins, Port C3 and C4 (Pin 23 and 24) to be connected to its slave (in this case MMC card).

4.4.1 Basic test without MMC card

This is an example of oversimplified system. SDO pin is connected directly to SDI pin. Consequently, anything transmitted on the SPI bus will also be received by the SPI receiver. This test shows the appropriate method to set up and use the SPI port.

C Codes available in Appendix A.

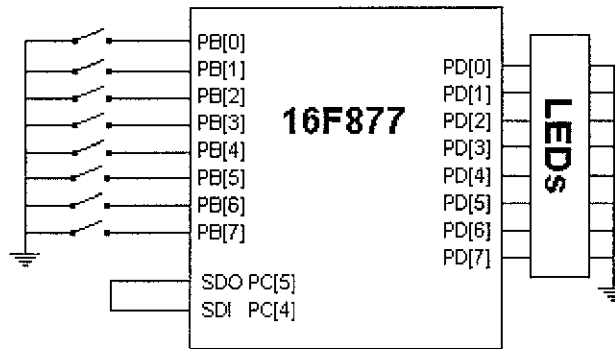


Figure 10 SPI Hardware Test

As a result from this test, what have been written in SDI through pull-ups switch is output to the LEDs through SDO. The built in functions are used to read and write data to the SPI port.

4.4.2 Initialize the MMC

The MMC requires initialization by SPI communication. This step is compulsory as part of the required initialization is to setup the address block to be transferred from MMC to the PIC. In the final stage, when the decoder chip (STA013 is connected), the respective data is then been transferred to the STA013 instead of PIC.

4.4.3 Connecting the MMC to the SPI

Basically, the connection between the Multimedia Card and SPI port consist of 4(four) physical pins as shown in Table 4. As been discussed earlier, SPI has a master-slave configuration; in this case, MMC will be the slave drive.

The output of PIC is a 5-volt tolerance and it's not match with the MMC, which is 3.3-volt tolerance. The simple way to interface a 5-volt output to the MMC input pins is with a series resistor that will limit the current when the 5-volt output is high. There is some input capacitance (3.5 pF) on the input pins, so adding a small capacitor in parallel with the resistor will allow the rapid edge to properly drive the input pin. The value of the resistor and capacitor are not critical, Figure 11 shows 4.7K and 47pF, which limits the steady-state current to less than 1/2 mA.

PIC Pins (Master)		MMC Pins (Slave)		Function
SDO	PC[5]	Data In	Pin 2	Data Out from PIC receive by MMC
SDI	PC[4]	Data Out	Pin 7	Data In to PIC sends by MMC
SCK	PC[3]	Clk	Pin 5	To synchronize clock signal for MMC to be the same as the PIC clock frequency
SS	PC[1]	CS	Pin 2	Slave Selection by PIC, to enable and disable the MMC for the SPI modes.

Table 4 Pin function between PIC and MMC

This circuit is applied on each output of the PIC (mainly SDO PC [5]) to the Data In (Pin 2) of MMC card. For the Data Out (Pin 7) of the MMC to the SDI PC [4] of the PIC, the pins do not require this step down circuit.

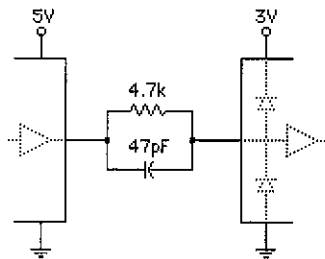


Figure 11 Interfacing 5-volt output to 3-volt tolerance input.

4.4.4 File transfer test with the present of MMC card

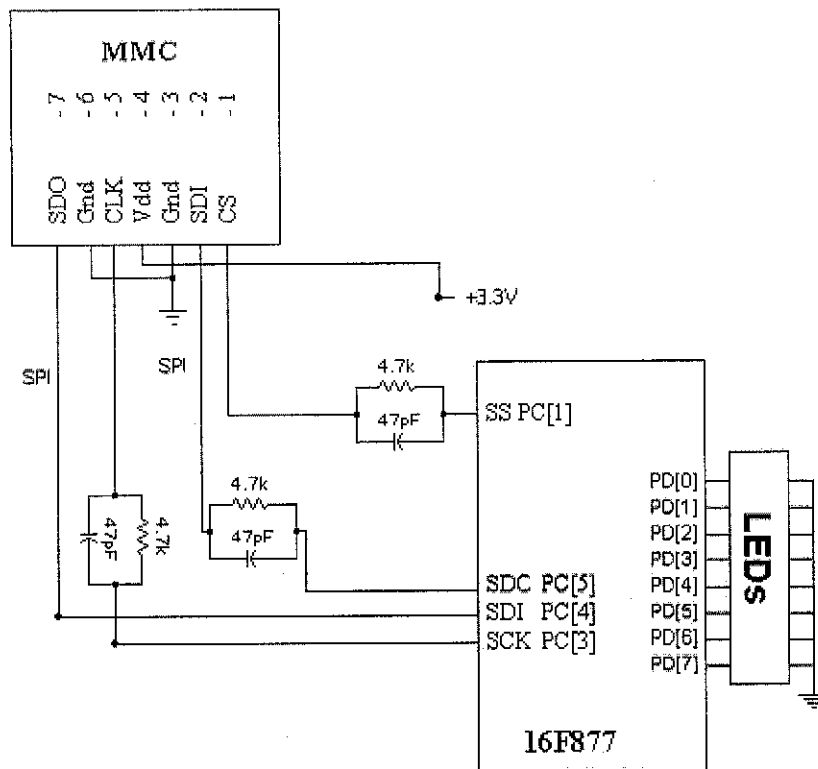


Figure 12 SPI Master (PIC) and Slave (MMC) communication.

As been discussed in early part of this chapter, SDO pin of MCU sent the address of data required, the Data Out pin of MMC (which is also SDO) sent the data to the SDI of the MCU.

As of above figure, if an array of 8 bits data has been stored in MMC been recalled, the received data will light up the LEDs according to what has been stored inside the MMC. For the ease of the troubleshooting, Intel-Hex file is being stored inside the MMC. This is to monitor the LED is light up corresponding what has been written in the HEX file.

4.4.5 Intel-Hex File

The Intel-hex format requires each line to begin with a “:” to denote it as a start of a line of a hex code. The first two digits indicate how many data bytes are on the line. In this file, the “20” in hexadecimal means there are actually 32 (in decimal) data

bytes on the line. The next two bytes (“8000”) denote the address of the first data byte in the line. The “00” indicates the nature of the data. Type 00 is simply regular data to be loaded, while type 01 is used to denote the last line in a hex file. The “FFFB30C0000004E4BD6618728001218BEE6B9E3001F00F1478904FCF3FDBFC0E” are the actual data and the last byte is the checksum, used for error checking.

```
:20800000FFFB30C0000004E4BD6618728001218BEE6B9E3001F00F1478904F
CF3FDBFC0E2F
```

Figure 13 Single line of Intel-Hex file

4.5 Send data

For the final stage of the design, the concept is to give the STA013 data when it requests more. The STA013 chip will determine the MP3 bit rate, consume the data at the correct speed, and give the request signal when it needs more. It also automatically detects the required sample rate (44.1 kHz, 48 kHz, etc) from the MP3 data and automatically adjusts the DAC clock. The speed to input the bits does not matter, as long as it's less than 20 Mbit/sec.

There are three signal lines used for sending the MP3 data to the STA013 decoder: SDI (DATA), SCKR (CLOCK), and DATA_REQ. The STA013 asserts the DATA_REQ line when more MP3 data is needed from the host controller. The host controller then feeds MP3 data, most significant bit first, on the SDI line. Each bit placed on SDI is clocked, by the controller, into the STA013 with the SCKR signal line.

4.5.1 PIC to MMC

Before the final output is investigated, it is important to make sure that the data is flowing from the MMC to the PIC; else the full connection to the STA 013 can not be troubleshot at ease. The software to tell microcontroller to send data to the MMC is written in C. Initially, the MMC is told to be ready in the SPI mode by the following lines (Figure 14).

```

int mmc_init()
{
    int i;

    SETUP_SPI(SPI_MASTER | SPI_H_TO_L | SPI_CLK_DIV_4 | SPI_SS_DISABLED);

    0*0x94 |= 0x40;           // set CKE = 1 - clock idle low
    *0x14 &= 0xEF;           // set CKP = 0 - data valid on rising edge

    OUTPUT_HIGH(PIN_C2);     // set SS = 1 (off)

    for(i=0;i<10;i++)       // initialise the MMC card into SPI mode by sending clks on
    {
        SPI_WRITE(0xFF);
    }

    OUTPUT_LOW(PIN_C2);     // set SS = 0 (on) tells card to go to spi mode when it receives reset

    SPI_WRITE(0x40);        // send reset command
    SPI_WRITE(0x00);        // all the arguments are 0x00 for the reset command
    SPI_WRITE(0x00);
    SPI_WRITE(0x00);
    SPI_WRITE(0x00);
    SPI_WRITE(0x95);        // precalculated checksum as we are still in MMC mode

    if(mmc_response(0x01)==1) return 1; // if = 1 then there was a timeout waiting for 0x01 from the mmc
}

```

Figure 14 Tell MMC to go to SPI modes

Then to output the content inside the MMC to Port D then light up the LEDs,

```

OUTPUT_LOW(PIN_C2);       // set SS = 0 (on)
SPI_WRITE(0x51);          // send mmc read single block command
SPI_WRITE(0x02*(varh));   // arguments are address
SPI_WRITE(0x00*(varh));
SPI_WRITE(0x02*(varl));
SPI_WRITE(0x00);
SPI_WRITE(0xFF);         // checksum is no longer required but we always send 0xFF
if((mmc_response(0x00))==1) return 1; // if mmc_response returns 1 then we failed to get a 0x00 response (affirmative)
if((mmc_response(0xFE))==1) return 1; // wait for data token

for(i=0;i<512;i++)
{
    output_D(~(spi_read())); // we should now receive 512 bytes
}

```

Figure 15 Output to Port D

4.6 Output

The output is connected to a speaker where a song is expected to be heard. However, since the input is less than 1 second, it is difficult to hear a song. Instead, only “bip” is heard. To overcome this, the output of CS4334 DAC is connected to an oscilloscope. The signal from the oscilloscope is compared with the signal produced before the program is run. If the decoder works, there should be difference in the output waveform.

4.7 Current circuit connection

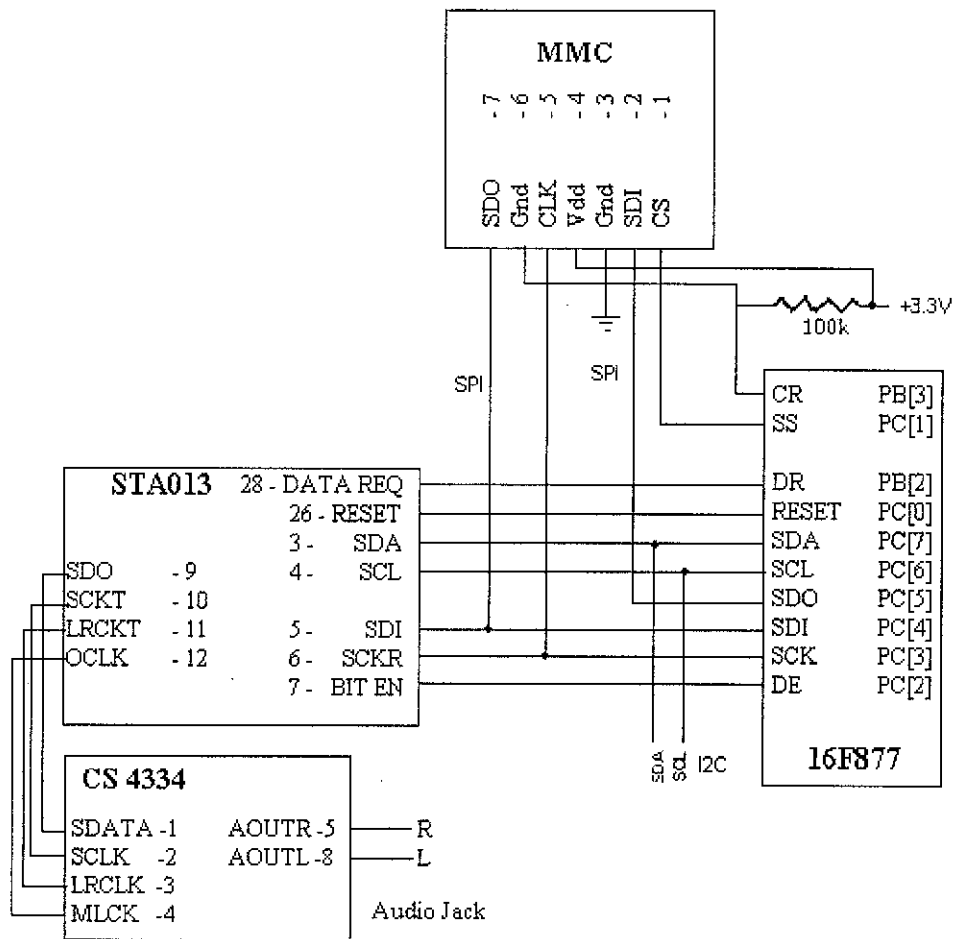


Figure 16 Current Circuit Connection

CHAPTER 5

CONCLUSION

The first requirement of this project is to build a basic knowledge on the microcontroller architecture and how to use it in many applications. The knowledge of programming in both C and assembly language is also required as to program the microcontroller. Enhancement in the programming can be done by reading materials and tutorials available on the microcontroller manufacturer websites.

Another requirement in this project is to know how to configure the MMC. This phase is the main concern in completing this project. If the decoder is not properly initialized, the whole decoding system will not function.

Physically, the basic portion of circuit in sending data from MMC to PIC has been setup for future enhancement. Communication between PIC and MMC is initialized for data transfer in both ways.

CHAPTER 6

RECOMMENDATION

The current project implements the basic applications for an MP3 player. In the future, to make sure the project progress is done systematically; the smallest possible module for each part can be developed. This is for the ease of troubleshooting.

MMC as the storage of MP3 files is quite troublesome for the end user. USB flash drive can be use later in project development.

Changes can be made to control volume, bass and treble via the I²C interface. A real MP3 data can also be used to replace the Intel-hex file. The output of the DAC can be connected to a headphone amplifier to provide proper output levels for headphones.

For future enhancement, more application can be designed and added to the player. An LCD can be introduced to display information to the user. It is also more convenient if there is wireless FM transmitter or more advance wireless communication protocol, Bluetooth, attached to the system.

REFERENCES

1. Microcontroller PICF877 Datasheet
2. Embedded C Programming for Microchip PIC, Barnett Cox & O'Cull, 2004
3. I²C Bus Technical Datasheet
4. <http://www.pjrc.com>
5. <http://www.microchip.com>
6. <http://ccsinfo.com>
7. <http://www.st.com>
8. <http://instruct1.cit.cornell.edu>
9. <http://www.leotronic.com>
10. <http://www.howstuffworks.com>
11. <http://www.terra.es>
12. <http://www.mcc-us.com>

APPENDICES

APPENDIX A CODES FOR SPI HARWARE TEST

```
#include <16F877.h>
#fuses HS,NOWDT
#use delay(clock=4000000)

void main()
{

    setup_spi(SPI_MASTER|SPI_L_TO_H|SPI_CLK_DIV_16);
    port_b_pullups(TRUE);
    spi_write(0x00);

    while (1)
    {
        if (spi_data_is_in())
        {
            output_D(~(spi_read()));
            spi_write(input_b());
        }
        output_low(PIN_C2);
        delay_ms(50);
        output_high(PIN_C2);
        delay_ms(50);
    }
}
```

APPENDIX B

GLOBAL.H

```
// Global.h
//
// defines port, tris & clock
//

#ifndef _GLOBAL_H
#define _GLOBAL_H

// globally define the delay
#define CLOCK_SPEED 14745600
#define delay(clock=CLOCK_SPEED)

// create desired types
#define bool short

// port defines
#define PORTA = 0x05
#define PORTB = 0x06
#define PORTD = 0x08
#define PORTE = 0x09

#endif
```

APPENDIX C

MMC2.H

```
// MMC2.h
// Library for initializing and reading from the MMC
//
// Pin configuration:
// -----
// PIC   MMC   MMC Pin#
// rA5   CS           1
// rC5   DataIn       2
//       GND           3
//       VDD           4   (3.3V)
// rC3   Clk           5
//       GND           6
// rC4   DataOut       7
// -----

#ifndef _MMC2_H
#define _MMC2_H
// -----
#include <global.h>
#include <address.h>
#include <STA013.h>
// -----

// slave select pin
#define MMCSS_PIN_C1

// MMC Commands
// Set block length for next read/write
#define MMC_CMD_SETLENGTH 0x50

// Write block to memory
#define MMC_CMD_WRITEBLOCK 0x54

// Read block from memory
#define MMC_CMD_READBLOCK 0x51

// Data token start byte
#define MMC_DATA_TOKEN 0xFE

// an affirmative R1 response (no errors)
#define MMC_R1_RESPONSE 0x00

// this variable will be used to track the current block length
// this allows the block length to be set only when needed
unsigned long _BlockLength = 0;

// error/success codes
#define MMC_SUCCESS 0x00
#define MMC_BLOCK_SET_ERROR 0x01
#define MMC_RESPONSE_ERROR 0x02
#define MMC_DATA_TOKEN_ERROR 0x03
#define MMC_INIT_ERROR 0x04
#define MMC_TIMEOUT_ERROR 0xFF

// -----
// MMC2Read
// -----
// Read data from the MMC to the MP3 decoder
// -----
// Arguments:
```



```

// Size = byte : max size of data to read
// address = Address32 : address to start reading from
//-----
// Returns: error or success code
//-----

#SEPARATE
unsigned int MMC2Read(unsigned long size, struct Address32 *address);

//-----
// _MMCSetBlockLength
//-----
// Sets the block length of the MMC (if needed)
//-----
// Arguments:
// length = unsigned long : new block length
//-----
// Returns: true if the block size was set successfully
// always true if no change made
//-----
// Notes:
// This is an internal routine and should not be called externally
//-----

#SEPARATE
bool _MMCSetBlockLength(unsigned long length);

// internal routine
// sends a command to the MMC

#SEPARATE

void _MMCSendCommand(unsigned int cmd, unsigned int arg1, unsigned int arg2, unsigned int arg3, unsigned
int arg4, unsigned int crc);

// internal routine
// waits for a valid response from the MMC and returns the response received

#SEPARATE
unsigned int _MMCWaitForResponse(unsigned int expected);

#endif

#endif

```

APPENDIX D

MMC.C

```
// MMC2.c
//
// Declares the constants and prototypes to interface to MMC
//

#ifndef _MMC2_C
#define _MMC2_C

//-----
#include <global.h>
#include <MMC2.h>
//-----

#ifdef _MMC2_C
unsigned int MMC2Read(unsigned long size, struct Address32 *address)
{
    unsigned long i = 0;
    unsigned int rValue = MMC_SUCCESS;
    bool done = false;

    // Set the block length to read
    if(_MMCSetBlockLength(size))
    {
        // SS = LOW (on)
        OUTPUT_LOW(MMCSS);

        // send read command

        _MMCSendCommand(MMC_CMD_READBLOCK, HIGH(address->upper),
            LOW(address->upper), HIGH(address->lower), LOW(address->lower), 0xFF);

        // check if the MMC acknowledged the read block command
        // it will do this by sending an affirmative response
        // in the R1 format (0x00 is no errors)

        if(_MMCWaitForResponse(MMC_R1_RESPONSE)==MMC_SUCCESS)
        {
            // now look for the data token to signify the start of
            // the data

            if(_MMCWaitForResponse(MMC_DATA_TOKEN)==MMC_SUCCESS)
            {
                // set the decoder to start reading data (pull high)

                OUTPUT_HIGH(STA013_BIT_ENABLE);
                // clock the actual data transfer, it will be read by the
                // decoder automatically since the decoder has been enabled
                // make sure that for each byte the decoder still wants data

                for (i=0; i < size; ++i)
                {

                    // wait until the decoder wants more data
                    while(!INPUT(STA013_DATA_REQUEST));

                    // read out the data
                    SPI_READ(0xFF);
                }
            }
        }
    }
}
#endif

```

```

// data done being streamed to decoder
OUTPUT_LOW(STA013_BIT_ENABLE);

// adjust the address based on the size of the transfer
AddressAdd(address, size);

// get CRC bytes (not really needed by us, but required by MMC)
SPI_READ(0xFF);
SPI_READ(0xFF);
}

    else
    {
        // the data token was never received
        rValue = MMC_DATA_TOKEN_ERROR;
    }
}

    else
    {
        // the MMC never acknowledge the read command
        rValue = MMC_RESPONSE_ERROR;
    }

// SS = HIGH (off)
OUTPUT_HIGH(MMCSS);

// give the MMC the required clocks to
// finish up what ever it needs to do
SPI_WRITE(0xFF);
}

    else
    {
        rValue = MMC_BLOCK_SET_ERROR;
    }
return rValue;
}

//-----
#endif

```

APPENDIX E

STA013.H

```
// STA013.h
//
// Declares the constants and prototypes to interface to the
// STA013 MPEG decoder
//

#ifndef _STA013_LIB_H
#define _STA013_LIB_H

#include <global.h>

// config file constants
#define STA013_CONFIG_START 0x0000
#define STA013_CONFIG_LENGTH 0x0FAE

// pin defines
#define STA013_BIT_ENABLE PIN_C2
#define STA013_DATA_REQUEST PIN_E2
#define STA013_RESET_PIN PIN_C0

// the address values required to be sent via IIC
// to the STA013 for register read and writes
#define STA013_ADDRESS_READ 0x87
#define STA013_ADDRESS_WRITE 0x86

// register addresses
// taken from STA013 documentation - pg 13-28
#define STA013_REGISTER_PLAY 0x13
#define STA013_REGISTER_RUN 0x72

// error/success codes
#define STA013_SUCCESS 0x00
#define STA013_CONFIG_ERROR 0x01
#define STA013_INIT_ERROR 0x02
#define STA013_REGISTER_WRITE_ERROR 0x04
#define STA013_NOT_PRESENT_ERROR 0x08

//-----
// _STA013RegisterWrite
//-----
// Writes a value to a register on the STA013
//-----

#SEPARATE
unsigned int _STA013RegisterWrite(unsigned int address, unsigned int value);

//-----
// _STA013RegisterRead
//-----
// Read a value from a register on the STA013
//-----

#SEPARATE
unsigned int _STA013RegisterRead(unsigned int address);

//-----
// STA013Init
//-----
```

```
// Initializes the STA013 MPEG decoder for use
//-----

#SEPARATE
unsigned int STA013Init();

//-----
// STA013Play
//-----
// Starts the decoder decoding
//-----

#SEPARATE
unsigned int STA013Play();

//-----
// STA013Stop
//-----
// Stops the decoder from decoding
//-----

#SEPARATE
unsigned int STA013Stop();

#endif
```

APPENDIX F

STA013.C

```
// STA013.c
//
//
// STA013 MPEG decoder library definition
//
// note: the "#use i2c..." statement must appear before this library is included
//

//if not define, then define
#ifndef _STA013_C
#define _STA013_C

//-----
#include <global.h>
#include <STA013.h>
//-----

#include <MMC2.c>
//-----

#SEPARATE
unsigned int _STA013RegisterWrite(unsigned int address, unsigned int value)
{
    // an ACK value is actually logical 0

    unsigned int rValue = STA013_SUCCESS;
    bool ok; // ok should return value 0

    // start the I2C communication
    I2C_START();

    // choose the STA013 device for writing
    // write, return 0

    ok = !I2C_WRITE(STA013_ADDRESS_WRITE); //value is 0x86(134)

    ok = !I2C_WRITE(address); //address is within the STA013 where we write the data

    ok = !I2C_WRITE(value); //value is the data passed from prog to sta013

    I2C_STOP();

    //check ACK bit returned by each byte transmit
    //return an error to main prog if no ACK bit received

    rValue = ok?STA013_SUCCESS:STA013_REGISTER_WRITE_ERROR;
    return rValue;
}

//-----

#SEPARATE
unsigned int _STA013RegisterRead(unsigned int address)
{
    unsigned int rValue = 0x0;
```

```

I2C_START();

I2C_WRITE(STA013_ADDRESS_WRITE); //read from 0x86(134)
I2C_WRITE(address); //this will pass the address to read function

I2C_START(); // start the read sequence
I2C_WRITE(STA013_ADDRESS_READ); //value is 0x87(135)

// get the register value - read byte from STA013
// will be returned to the main program

rValue = I2C_READ(0); //reads are not ack (0)

// stop the transmission
I2C_STOP();

// return our read value
return rValue;
}

//-----
#SEPARATE
unsigned int STA013Init()
{
    unsigned int rValue = STA013_SUCCESS;
    unsigned long i;
    unsigned int address;
    unsigned int value;
    bool ok;

    // reset the hardware
    OUTPUT_LOW(STA013_RESET_PIN);
    delay_ms(100);
    OUTPUT_HIGH(STA013_RESET_PIN);

    // give chip a software reset
    _STA013RegisterWrite(16, 1);

    // check the sta013, read from add 0x01
    //return ACK of 0xAC to 0x01, if no ACK - no STA013 present

    if(_STA013RegisterRead(0x01)!=0xAC)
    {
        rValue == STA013_NOT_PRESENT_ERROR;
    }

    //-----
    //this section is reserved for the configuration file (if needed)
    //send the config BIN file that is stored in the MMC (tentatively)
    //use STA013 write function & should get ACK for each write

    for(i = 0; rValue == STA013_SUCCESS && i < STA013_CONFIG_LENGTH;i+=2)
    {
        //read the register address
        address = MMC2Read(STA013_CONFIG_START+i);

        //read the value
        value = MMC2Read(STA013_CONFIG_START+i+1);

        //write the value to the register
        if(_STA013RegisterWrite(address, value)!=STA013_SUCCESS)
        {
            //there was an error streaming the config file
            rValue = STA013_CONFIG_ERROR;
        }
    }
}

```

```

        }

//insert a small delay after a write to the soft reset register
if(address==16)
{
    delay_ms(1);
}
}

//-----

if(rValue == STA013_SUCCESS)
{

// setup for 24-bit output to the CS4334 DAC

ok = _STA013RegisterWrite(84, 1);
ok |= _STA013RegisterWrite(85, 33);

// PLL setup for 14.850 Mhz Clock
// and 256 Oversampling rate
// computed using ConfigPLL v1.0
//(http://us.st.com/stonline/prodpres/dedicate/mp3/sw/cpll.exe)

ok |= _STA013RegisterWrite(6, 11);
ok |= _STA013RegisterWrite(7, 0);
ok |= _STA013RegisterWrite(11, 3);
ok |= _STA013RegisterWrite(80, 16);
ok |= _STA013RegisterWrite(81, 147);
ok |= _STA013RegisterWrite(82, 236);
ok |= _STA013RegisterWrite(97, 14);
ok |= _STA013RegisterWrite(100, 129);
ok |= _STA013RegisterWrite(101, 105);
ok |= _STA013RegisterWrite(5, 161);

// this was NOT found in the data book specifically
// but is thought to be required to work
// this will enable the data request pin
// enable the data request pin and make sure it has
// polarity such that high indicates data request

ok |= _STA013RegisterWrite(24, 4);
ok |= _STA013RegisterWrite(12, 1);

// make sure that the decoder is not muted
ok |= _STA013RegisterWrite(0x14, 0);

// sta013 run
ok |= _STA013RegisterWrite(STA013_REGISTER_RUN, 1);

rValue =(ok==STA013_SUCCESS)?STA013_SUCCESS:STA013_INIT_ERROR;

return rValue;
}
}

//-----

#SEPARATE
unsigned int STA013Play()
{
// start the decoding process

return _STA013RegisterWrite(STA013_REGISTER_PLAY, 1);
}

//-----

```



```
#SEPARATE
unsigned int STA013Stop()
{

// stop the decoding process
return _STA013RegisterWrite(STA013_REGISTER_PLAY, 0);
}
//-----

#endif
```

APPENDIX G

PIC.H

```
// PIC.h
//

// define retry counts for various actions
#define STA013_INIT_RETRY_COUNT 5

// initializing
#define stInitializing 0

// stopped, not playing
#define stStop 1

// playing songs
#define stPlay 2

// error initializing STA013
#define stSTA013InitError 6

// error setting value on STA013
#define stSTA013Error 7

//-----
// Global Data members
//-----

unsigned int _CurrentState = stInitializing; // default to initializing state

// these are made global as they are used frequently and this saves on RAM usage
//unsigned int retry;
//unsigned int val;

//-----
// Process
//-----
// streams data to the decoder when the
// system is playing and handles all
// automatic track advances
//-----

#INLINE
void Process();

//-----
// SetSystemState
//-----
// sets the currnet system state and
// displays a message if relavent
//-----

#SEPARATE
void SetSystemState(unsigned int state);
```

APPENDIX H

PIC.C

```
#include <16F877.h>
#fuses HS,NOWDT,NOBROWNOUT,NOPUT,NOPROTECT
#use i2c(master, sda=PIN_C7, scl=PIN_C6)
#include <pic.h>
#include <global.h>
#include <MMC2.c>
#include <STA013.c>
//-----
#include <string.h>
//-----

void output(void);
char swap_bits(char test);

void main()
{
    unsigned int retry;
    unsigned int val;

    // the STA013 MPEG decoder
    retry = STA013_INIT_RETRY_COUNT;
    do
    {
        val = STA013Init();
        --retry;
    }
    while(val!=STA013_SUCCESS && retry > 0);

    if(val==STA013_SUCCESS)
    {
        Process();
    }
}
//-----
#INLINE
void Process()
{
    unsigned int i=0;
    unsigned long ptr=0;
    unsigned long data;

    const char hexdata[128] = {0xFF,0xFB,0x30,0xC0,0x00,0x00,0x04,0xE4,0xBD,
    0x66,0x18,0x72,0x80,0x01,0x21,0x8B,0xEE,0x6B,0x9E,0x30,0x01,0xF0,0x0F,0x14,
    0x78,0x90,0x4F,0xCF,0x3F,0xDB,0xFC,0x0E,0xFF,0xEC,0xC7,0xFF,0xE0,0x74,0x21,
    0x09,0xFF,0xE4,0x3E,0x7F,0xFF,0xEC,0x07,0x78,0x61,0xDF,0xE9,0x82,0x0E,0x27};

    for (i=0; i < 128; i++) // clock the actual data transfer and receive the bytes
    {
        data=hexdata[ptr++];
    }
}
```

```

// set the decoder to start reading data (pull high)

    OUTPUT_HIGH(STA013_BIT_ENABLE);

// clock the actual data transfer, it will be read by the
// decoder automatically since the decoder has been enabled
// make sure that for each byte the decoder still wants data

    for (i=0; i < 128; ++i)
    {
        // wait until the decoder wants more data

        while(!INPUT(STA013_DATA_REQUEST));

        // read out the data

        data;
    }

    OUTPUT_LOW(STA013_BIT_ENABLE);

    output(); //to blink led
}

//-----
void output()

//led blinking indicates pic is working

{

    set_tris_b(0x00);          //to set all pins as output, 1 as input

    while(1)

    {

        PORTB=0x18;           //blink at port b4 and b5

    }

}

```