# MSP-432 Based MP3 Player Using Bluetooth

FINAL PROJECT REPORT

ECEN 5613 EMBEDDED SYSTEMS DESIGN

DECEMBER 14, 2019

Project Members:

Sharan Arumugam

Atharv Desai

# Table of Contents

# Acknowledgements

We would like to express our gratitude to Prof. Linden McClure for illuminating us with precious guidance throughout the project path. We would like to thank the TA's for helping us to debug the problems and suggesting us the fallback plans.

We would like to thank the Electrical Engineering department and specially the Embedded Systems Design lab for providing us the Oscilloscopes and other resources to work with. We would like to thank Lauren Ma'am from ITLL Electronics Fabrication lab for teaching us DIP chip soldering.

# 1. INTRODUCTION

We feel that the intrinsic motive of an embedded systems engineer is to design products which can improve or improvise the daily lives of people and make a positive difference in their comfort. Music plays a role making human lives better and so we wanted to design an embedded system which would be revolving around music. This motivation gave us the idea of making a wireless MP3 Player.

While working on these projects, we worked on concepts such as FAT storage systems, MP3 and WAV files decoding as well as wireless communication through Bluetooth. We also delved into understanding audio amplification and SNR (Signal to Noise Ratio) while working with Bone Conduction Transducer and 8 Ohm Speaker. During this project, we acquired in-depth knowledge of communication protocols like UART, SPI interfaces while writing the firmware for it and hardware such as MP3 decoders and audio amplifiers, in order to build systems with the MSP-432.

## 1.1 System Overview

Our project can be bisected into two approaches. These two approaches are based on the hardware used to fetch the WAV files from SD card and playing using different MP3 decoder boards.

The System using first approach consists of following hardware components:

1. MSP-432 board
2. VS1053 MP3 Decoder Module
3. SD card Shield
4. HC-06 Bluetooth Module
5. Audio Amplifier and Speaker Circuitry

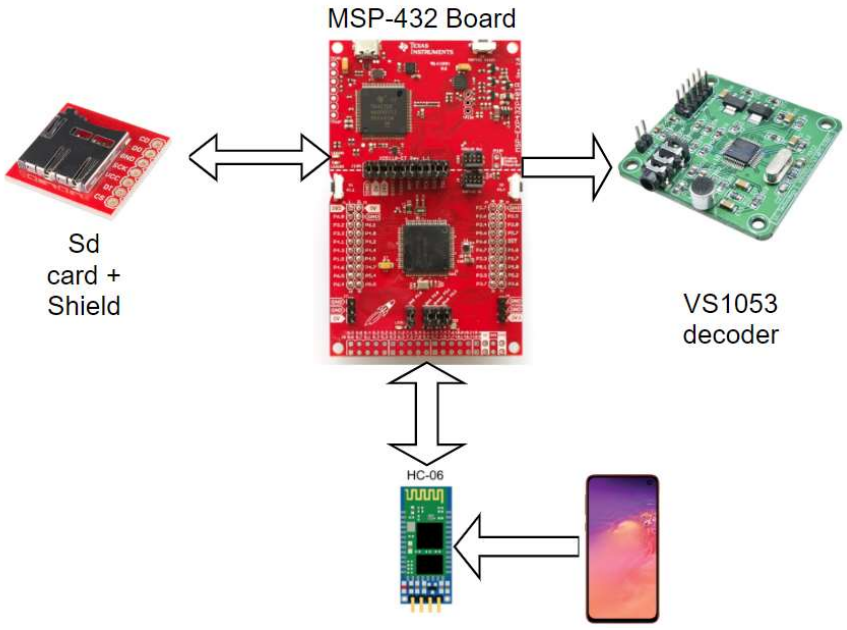



*Figure 1: System overview for First Approach*

Because of some issues which we encountered in the hardware of the VS1053 board, we had to switch to WTV020 MicroSD Audio Module to fetch and decode MP3 files.

The System using second approach consists of following hardware components:

1) MSP-432 board
2) WTV020 SD MicroSD Audio Module
3) HC-06 Bluetooth Module
4) Audio Amplifier and Speaker Circuitry

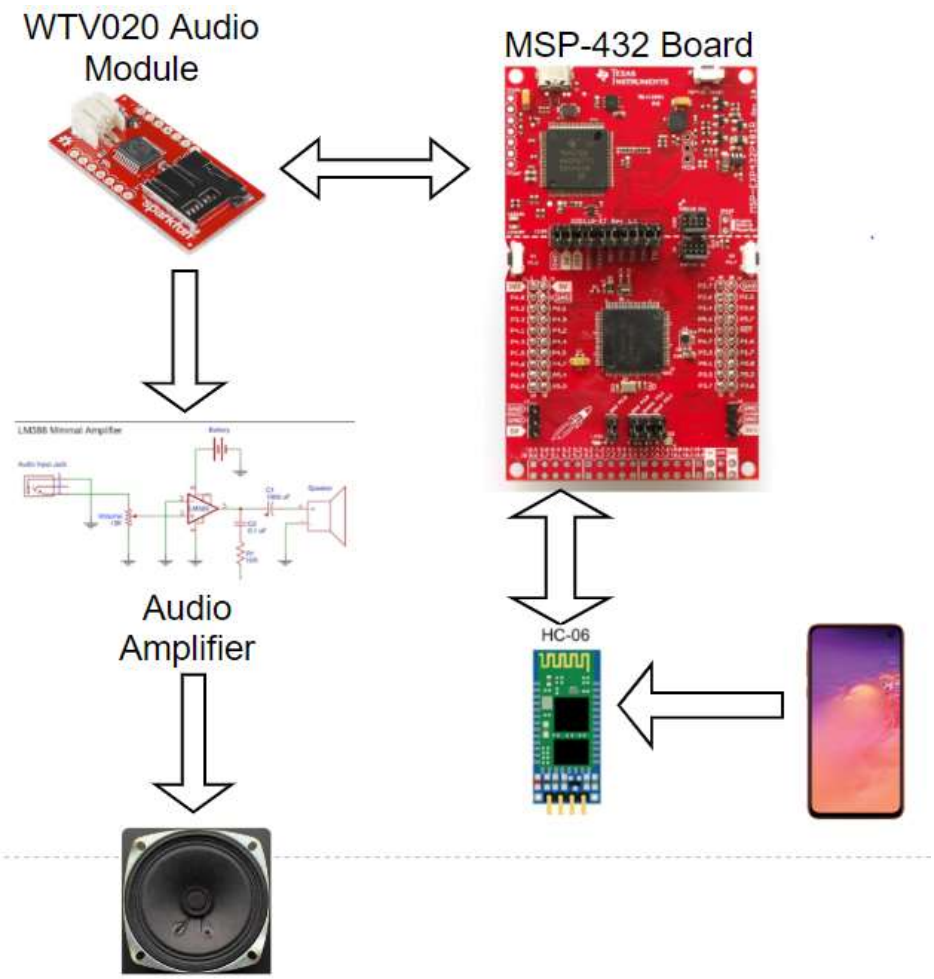



*Figure 2: System overview for Second Approach*

## 1.2 Task Allocation

| TASKS | ACCOMPLISHED BY |
|---|---|
| VS1053 Interfacing with MSP-432 (using SPI protocol) | Sharan Arumugam |
| SD Card Interfacing with MSP-432 (using SPI protocol) | Atharv Desai |
| WTV020 MicroSD Audio Module Interfacing with MSP-432 (using Bit Banging) | Sharan Arumugam |
| HC-06 Interfacing with MSP-432 (using UART protocol) | Atharv Desai |

| | |
|---|---|
| Design of Audio Amplifier Circuitry<br>(using LM386 circuit and 8 ohm Speaker) | Atharv Desai |
| LCD Optrex 20434 Interfacing with MSP-432 | Sharan Arumugam |
| Testing & Debugging the overall system | Both |

# 2. TECHNICAL DESCRIPTION

This section will give the overall view of the board designs for the two approaches in section 2.1, newly introduced hardware components in the project in section 2.2, the explanation for the firmware written for interfaces in section 2.3, the flow of the code in the software section 2.4 and testing & debugging process in section 2.5.

## 2.1 Board Design

### 2.1.1 SD card and VS1053 Module Based System

Shown below is the schematic of the board design using first approach in Figure 3 and the actual hardware setup in Figure 4.

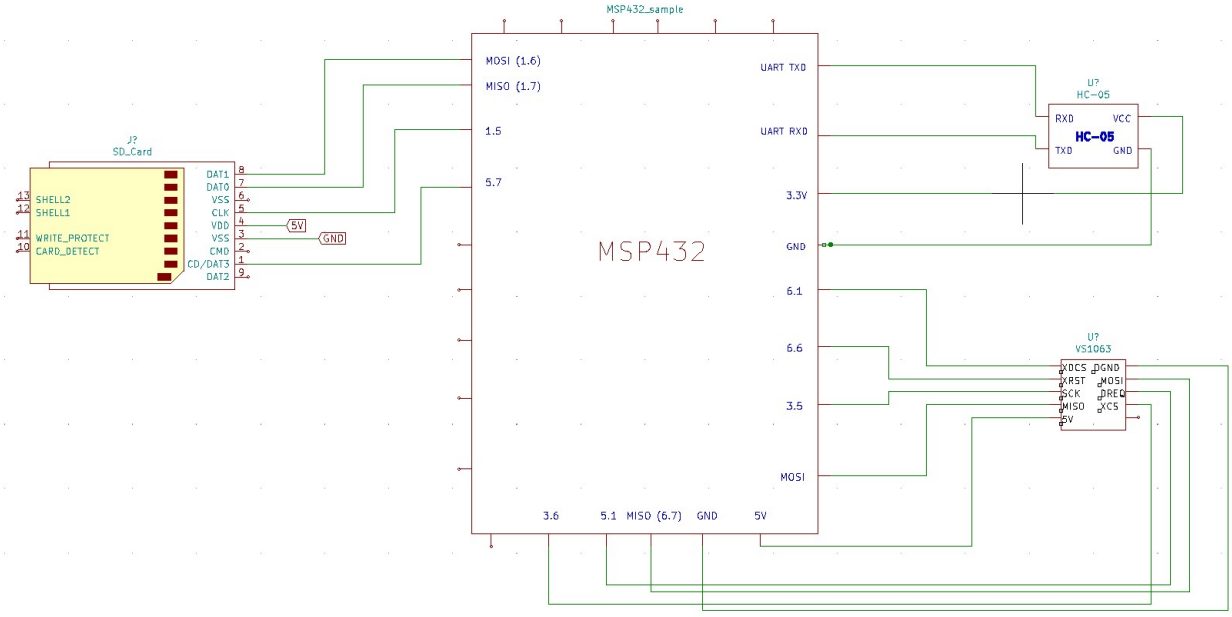



*Figure 3: Schematic for board design using First Approach*

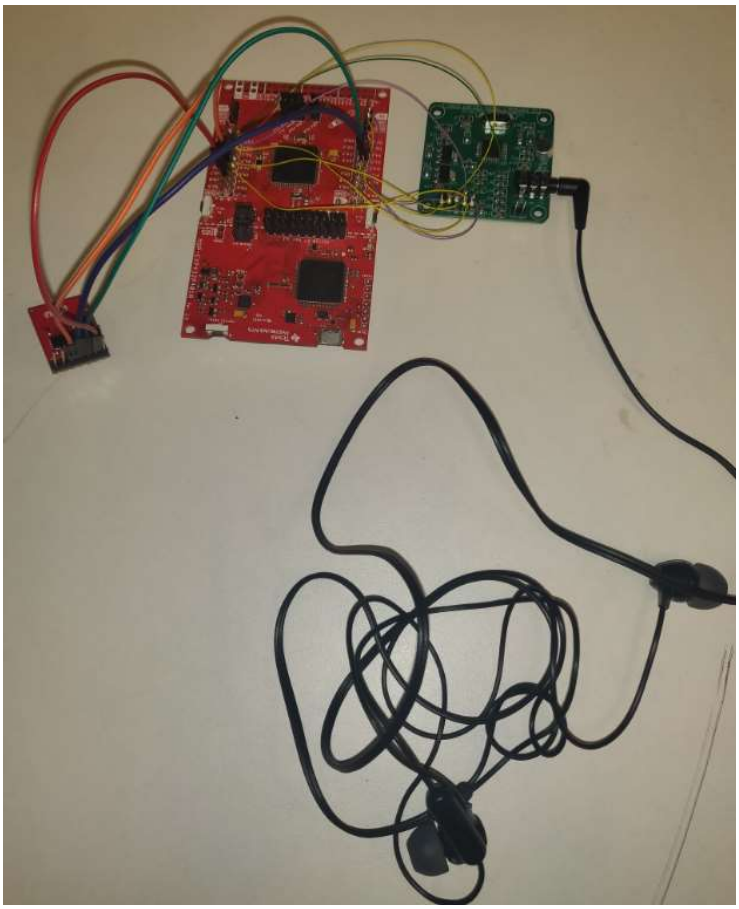


*Figure 4: Hardware Setup of System using First Approach*

Due to the issues in the breakout board of the VS1053, though the data was being transferred accurately to the board, we weren't receiving the audio through the audio jack. Therefore, we switched to the second approach which is demonstrated below in the form of hardware schematic and the actual setup in section 2.1.2

### 2.1.2 WTV020SD AUDIO MODULE & BLUETOOTH BASED SYSTEM

Shown below is the schematic of the board design using first approach in Figure 5 and the actual hardware setup in Figure 6.

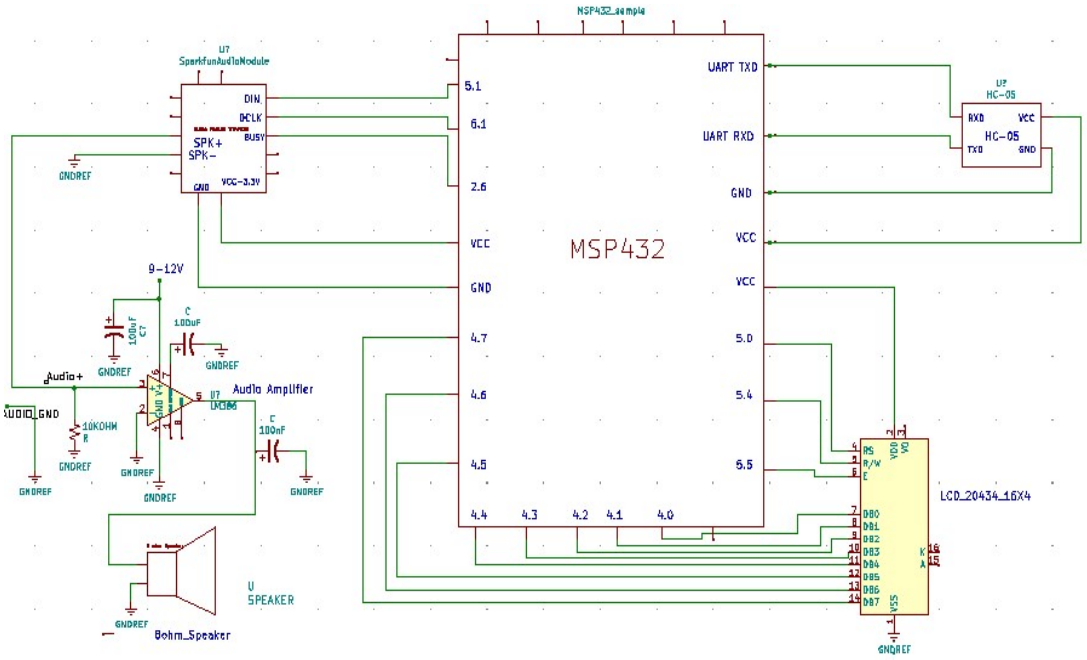



*Figure 5: Schematic for board design using Second Approach*

The actual hardware setup has been demonstrated in the photo shown below.

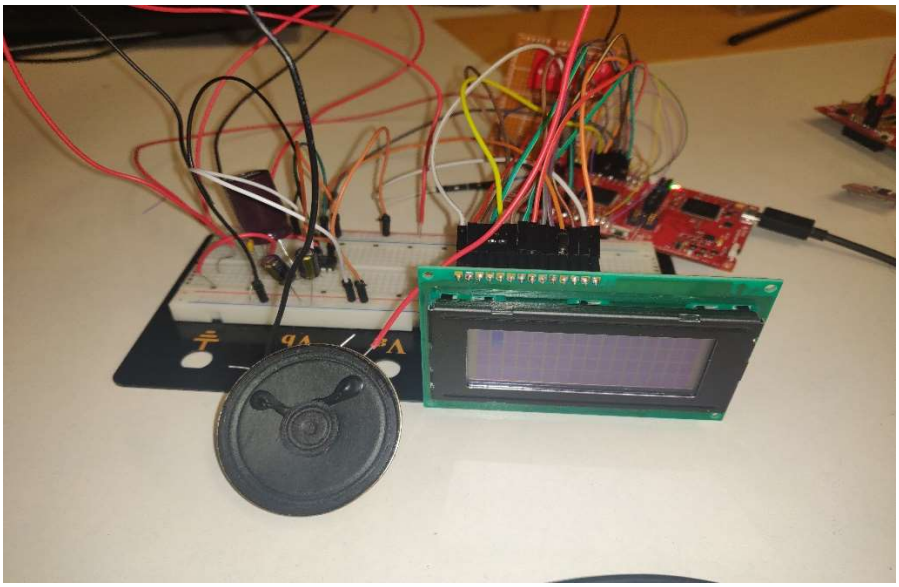


Figure 6: Hardware Setup of System using Second Approach

## 2.2 New Hardware Description

This section will provide a synopsis of all the new hardware components and their specifications, i.e. the VS1053 Module and its breakout board architecture in 2.2.1, the SD card adaptor module in 2.2.2, WTV020 MicroSD Audio Module and the description of its pins in 2.2.3, HC-06 Bluetooth Module and its functionality in 2.2.4, and finally the Audio amplifier and speaker circuitry using LM386 in section 2.2.5.

### 2.2.1 VS1053 Module

VS1053 audio decoder [1] is a low power DSP processor core containing chip with 16KB of Internal RAM, 0.5KB of Data RAM, serial control and input data interfaces as well as ADC and DAC for mic, stereo and an earphone amplifier. The block diagram of the VS1053 breakout board is shown below.

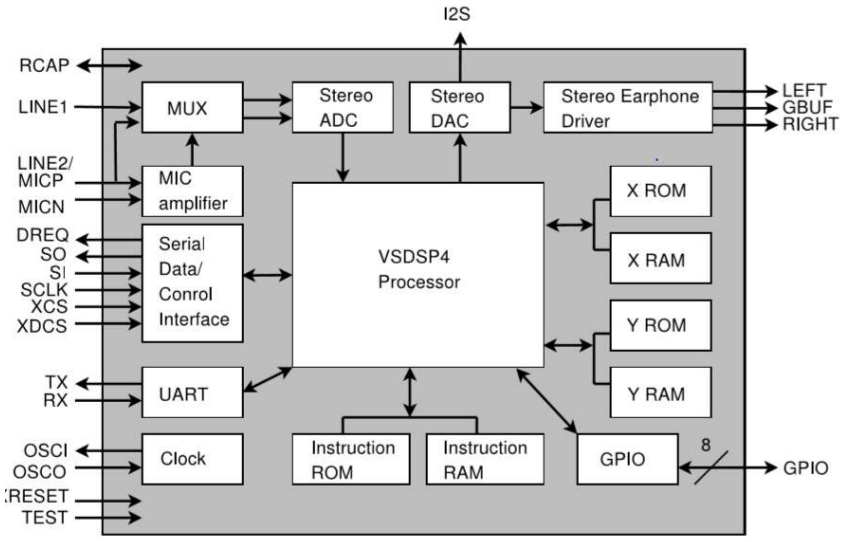



Figure 7: VS1053 Board Block Diagram

The functionality of some of its important pins has been described below:

i. DREQ (Data Request Pin) - The DREQ pin is used indicates whether the 2048-byte FIFO in VS1053 can receive data. When DREQ pin is set high, VS1053 can hold 32 bytes of SDI data or one SCI command where SDI stands for Serial Data Interface (SDI) and Serial Command Interface (SCI) .When DREQ is low, it indicates that the stream buffer is full for the duration of an SCI command. Because of the 32-byte safety area, VS1053 becomes compatible with low speed microcontrollers.

ii. SCK (Serial Clock Input) - The serial clock can work internally as the master clock and can be gated or continuous.

iii. SI (Serial Input) - On the rising SCK edge and if XCS is low, sampling of SI takes place

iv. SO (Serial Output) – On falling clock edge, SO helps to get serial output data while in writing mode, the pin is in high impedance state.

v. MISO & MOSI – These are the SPI protocol pins which stand for Master IN Slave Out and Master Out Slave In respectively.

While interfacing the VS1053 with MSP- 432, we used SCI and SDI for read and write functions. The functions of these two pins has been described below.

i. SCI (Serial Control Interface) – It allows 16-bit data transfer and its reading and writing registers are used for several operations like loading user programs, obtaining encoded data, getting status data and controlling clock and other parameters.

ii. SDI (Serial Data Interface) – It allows the transfer of compressed decoder's data and the output is low if invalid analog inputs are given.

In the project, we used the breakout board as shown in Figure 8. Earlier, we tried to design a custom board on our own, but we faced some issues which will be explained in the section issues and error analysis further.

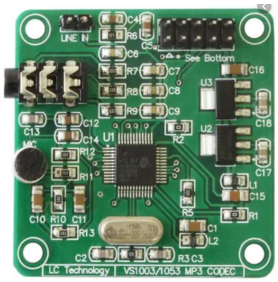



*Figure 8: VS1053 Breakout Board*

### 2.2.2 SD card Adapter Shield

The SD card acts as an external storage device to store the WAV or MP3 files and therefore, interfacing of SD card shield with MSP-432 plays an integral role in our project. We used the Sparkfun MicroSd Transflash Breakout shield [2] to interface the SD card with MSP-432. This module was interfaced using SPI protocol and the functionality of the pins with special features on the board has been explained below.

i. DO (Data Output) - This pin acts as MOSI (Master In Slave Out) pin and is connected to the MISO pin of the MSP-432. It is responsible for data transfer from SD card to MSP-432 while fetching the WAV files to the decoder.

ii. DI (Data Input) - This pin acts as MISO (Master Out Slave In) pin and is connected to the MOSI pin of the MSP-432. It is responsible for data transfer from MSP-432 to SDcard.

iii. SCK ( Serial Clock) - This pin is connected to Clock signal 1.5 pin on MSP432 for clock synchronization.

iv. CD (Card Detect) - This pin acts as a flag to check whether the SD card is inserted properly in the shield or not.

v. CS (Chip Select) - This is an active low pin which can also be called as Slave Select (SS) which is used to select one of the multiple slaves to transfer the data.

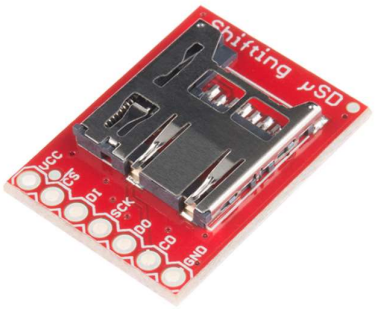



Figure 9: SD Card Shield

### 2.2.3 WTV020 MICROSD AUDIO MODULE

This module consists of SD card slot and WTV020 decoder chip in the breakout board. It works at the sampling rate ranging from 6 to 36 Khz [3]. The On-board SD card adapter can be used to store the WAV or MP3 files. The application diagram and its explanation for this module is shown below.

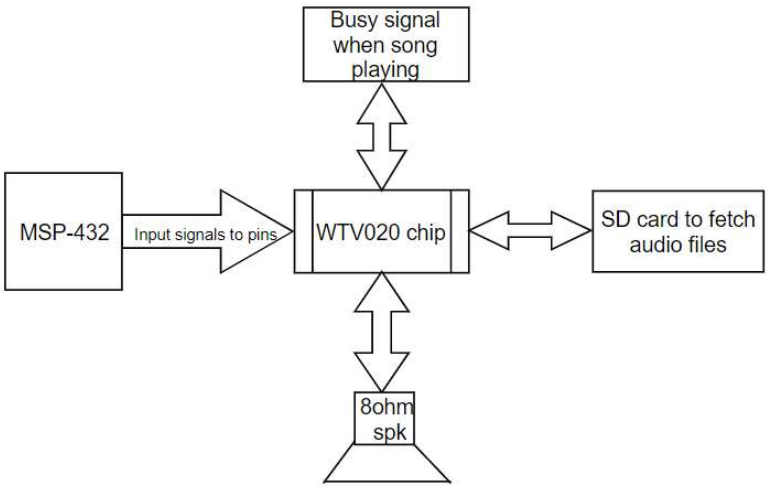



*Figure 10: Application Diagram for MicroSD audio Module*

As observed in the Application Diagram, WTV020 chip receives the control signals from the controller like MSP-432. Based on the control signals received at its input pins, it retrieves the audio files from the SD card, decodes the file and feeds it as an output to the speaker or to the audio amplifier circuitry. The functions of some of the important pins in the MicroSD Audio Module are

i. BUSY pin: This pin is high when the song is being played else it is set low.
ii. DOUT pin: Data input which will be received by the microcontroller.
iii. DCLK pin: Microcontroller or MSP-432 will provide the clock signal through this pin.
iv. RESET pin: provides 6ms of pull down to the whole module to reset it.
v. SPK +/-: provides the audio output signal to the speaker or audio amplifier.

Figure 11 below shows the actual breakout board of the WTV020 MicroSD Audio Module.

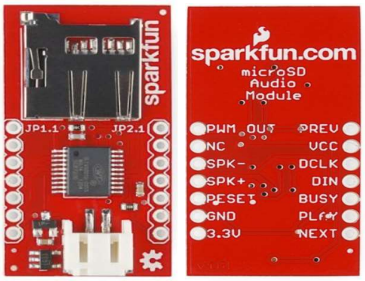



*Figure 11: MicroSD audio Module Breakout Board*

### 2.2.4 HC-06 BLUETOOTH MODULE

HC-06 Bluetooth Module [4] works on Bluetooth V2.0 Communication Protocol. It acts only in slave mode and works its data transfer speed (BT speed) is 2.1 Mb/s. The functionalities of the pins essential for communication are shown below.

i. Key pin: If this pin is set high, module works in AT command mode and if its set low, it works in normal Bluetooth mode. Since HC-06 works only in slave mode, it doesn't require AT commands and so HC-06 breakout board doesn't have this pin.

ii. TxD pin: This pin is used to the transfer the data from the Bluetooth module to the microcontroller. It works at default baud rate of 9600 bps. In our project, we used UART communication protocol to configure the TxD pin as well as the RxD pin.

iii. RxD pin: This pin receives the input data serially from the microcontroller to the Bluetooth module. Its default baud rate is of 9600 bps.

iv. State pin: Connected to LED and it shows whether the HC-06 is connected or not. HC-06 is unconnected if the LED is in the blinking state. Constant LED ON indicates the HC-06 is connected.

The figure below shows the breakout board of the HC-06 and its placement on the perfboard.

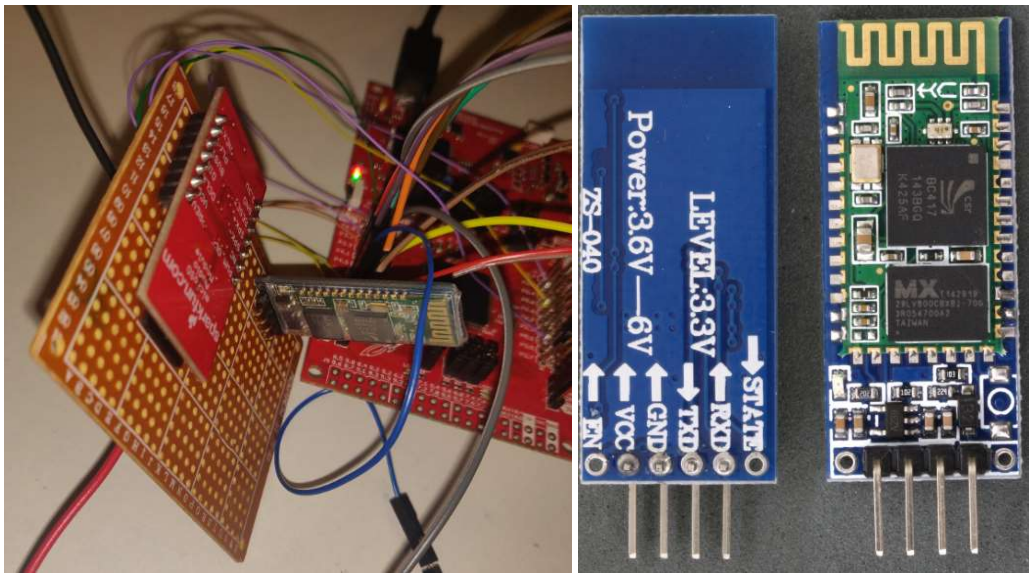



*Figure 12: HC-06 Placement on the perfboard*

Also, we are using Bluetooth Terminal Android App to send the ASCII values from mobile to the HC-06 module by setting up the buttons to ASCII values (e.g ON button set to ASCII value 1 in App GUI on figure 13).

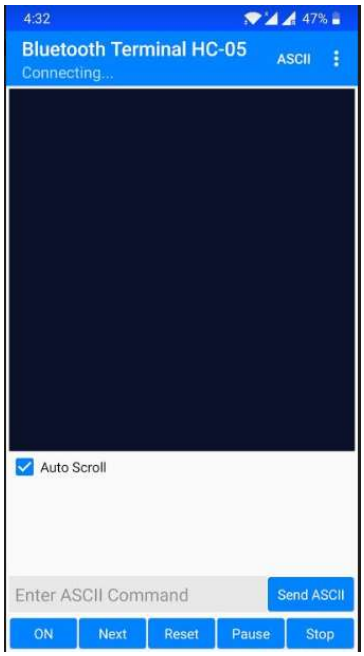



*Figure 13: Bluetooth Terminal App GUI*

### 2.2.5 LM386 AUDIO AMPLIFIER & SPEAKER CIRCUITRY

The output from the MicroSD Audio Module was giving a low power audio output. Therefore, to amplify the audio output, we implemented the audio amplifier circuitry using LM386 IC. LM386 is a low voltage audio power amplifier [5] which operates in the range of 5 to 18 V and its gain can be varied from 20 to 200. Varying the potentiometer values between 1 and 8 pin will change the gain accordingly. While pin 7 acts as a bypass to connect directly to signal input to avoid noise amplification.

The audio amplifier circuitry using LM386 with the 8ohm Speaker at the output has been demonstrated below.

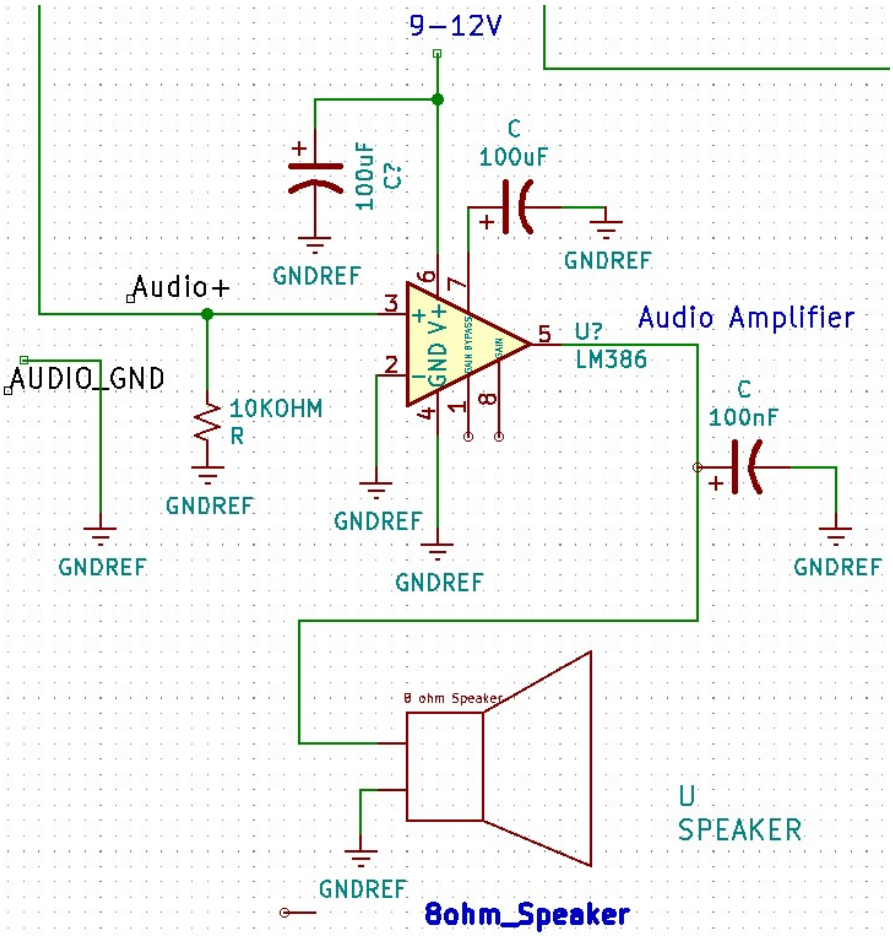



*Figure 14: Schematic block for LM386 Audio Amplifier Circuitry*

The resistor connected on pin 3 acts as a pull-down resistor and it helps to drive the input to GND when the source isn't connected to avoid noise.

The output from pin 5 if fed along with two capacitors connected in parallel to ground. These capacitors serve the purpose of filtering high frequency noise and for smoothing the output.

The actual breadboard setup has been demonstrated in the figure 14. It consists of the auxiliary capacitor and resistor values implemented on the breadboard as shown in schematic.

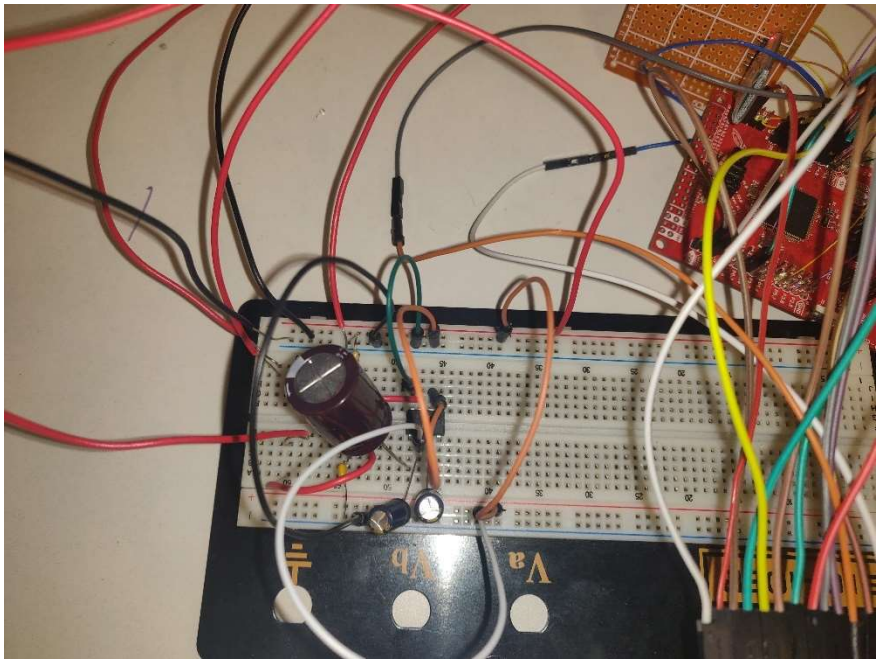


*Figure 15: Breadboard Implementation for LM386 Audio Amplifier Circuitry*

## 2.3 FIRMWARE DESIGN FOR COMMUNICATION PROTOCOL

Around 1150 lines of Code were written for the project.

**SPI Setup on the MSP432**

In our project, MSP432 is setup to be the master in every SPI transaction. SPI is conducted on the microcontroller using the EUSCI-B module which have a dedicated SPI peripheral which excuses us from developing a bit-banging SPI driver.

The control register CTLW0 allows to select the SPI mode, clock source and Pre-scaler settings, choice of master device and size of data as detailed on the user manual of the MSP432 [9].

Clock and prescaler values can be calculated using TI's online calculator which automatically gives you the values to be loaded for a desired data rate according to the selected clock frequency.

```
void spi_init()
{
            // Configure SPI for port 3 , other port for SD card
            EUSCI_B2->CTLW0 = 0;
            EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_SWRST; // Put eUSCI in reset
            EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_MST; //MSP432 in master mode
            EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_MSB; //MSB first mode
            EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_SYNC; //Synchronous
            EUSCI_B2->CTLW0 |= (EUSCI_B_CTLW0_CKPH); // Clock phase bit set high as per datasheet requirements
            EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_SSEL__SMCLK|EUSCI_B_IE_RXIE0; //Clock source selected as ACLK and receive interrupt enabled
            EUSCI_B2->BRW = 6; //This divides output frequency, by 6,

            //Configure GPIO pins to act as spi bus

            P3->SEL0 |= BIT5 | BIT6 | BIT7;
            P3->SEL1 &= ~(BIT5 | BIT6 | BIT7);
            //restart the state machine
            EUSCI_B2->CTLW0 &= ~(UCSWRST);
            //restart the state machine

            __enable_irq();
   // Enable eUSCI_B2 interrupt in NVIC module
   NVIC->ISER[0] = 1 << ((EUSCIB2_IRQn) & 31);

}
```

*Figure 16: SPI initialisation*

Received data is stored in the peripheral's RXBUF register and data to be transferred is loaded into the peripheral's TXBUF register. Depending upon whether an interrupt driven/ polling driven driver, changes can be made in the control register to enable the interrupt.

### 2.3.1 MCU INTERFACING WITH VS1053 (SPI PROTOCOL)

The VS1053B board supports its control interface through two protocols, UART and SPI. We chose to work with SPI as neither of us had worked with it in any of our assignments and we

also felt that SPI drivers would allow us to have better control over the transfer speed. The driver consists of 3 functions: an initialisation function, spi_transmit function and spi_read function.

Even though the board used an underlying SPI interface, it has two independent chip select pins which had to be toggled independently to indicate whether a COMMAND read/write or DATA write weas taking place.

The board does support multiple modes [1] for the chip select functionality. We use mode 1, which requires separate toggling of the chip select pin whereas there does exist another mode which allows one pins inverted output to be applied to the other i.e, if XCS (Command Select) is high then XDCS is set low. This would make XCS as the sole chip select pin which needs to be toggled but in order to retain more control over and to make debugging easier, we chose to set the chip selects individually.

**COMMAND WRITE**

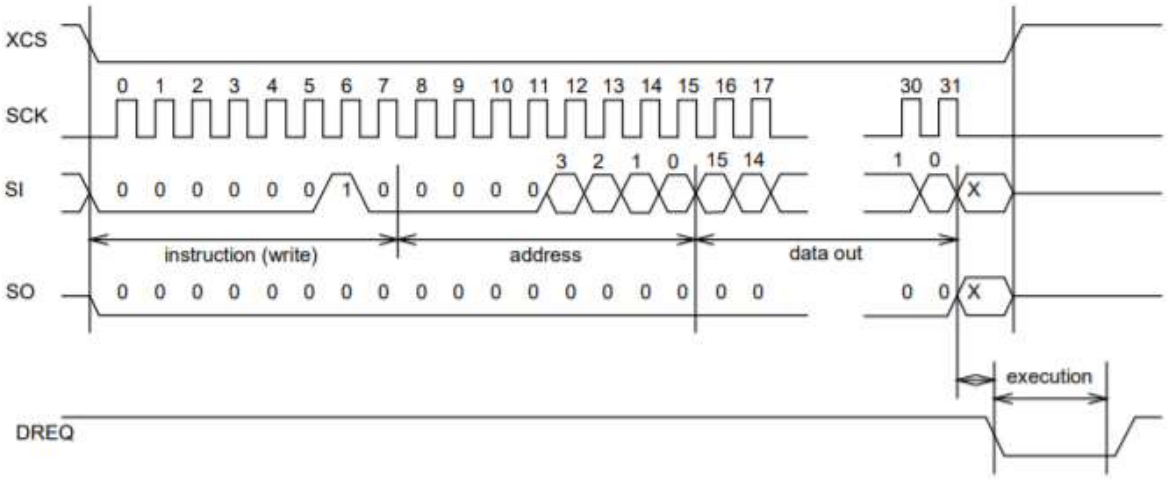



*Figure 17: Write Operation: Taken from VS1053 datasheet. XCS refers to command select pin DREQ indicates whether chip is ready to receive next operation.*

The XCS pin is set to be pulled low during the entire SPI transaction, serving as sort of an active low chip select. SCK, SI and SO are the SPI pins set to be used with the VS1053 while the DREQ pin is entirely left to be controlled by the chip itself and is set as input on the microcontroller. After a transaction is completed, the status of DREQ pin is read and execution is halted till it switches back to high state.

In the firmware code written, the COMMAND write was tested to work properly as it is successfully able to pass the software tests posed by the initialisation sequences as provided in the coding example provided by the VS1053 APPLICATION NOTE:PLAYBACK and RECORDING provided on the manufacturer's (VLSI solutions) website.

**COMMAND READ**

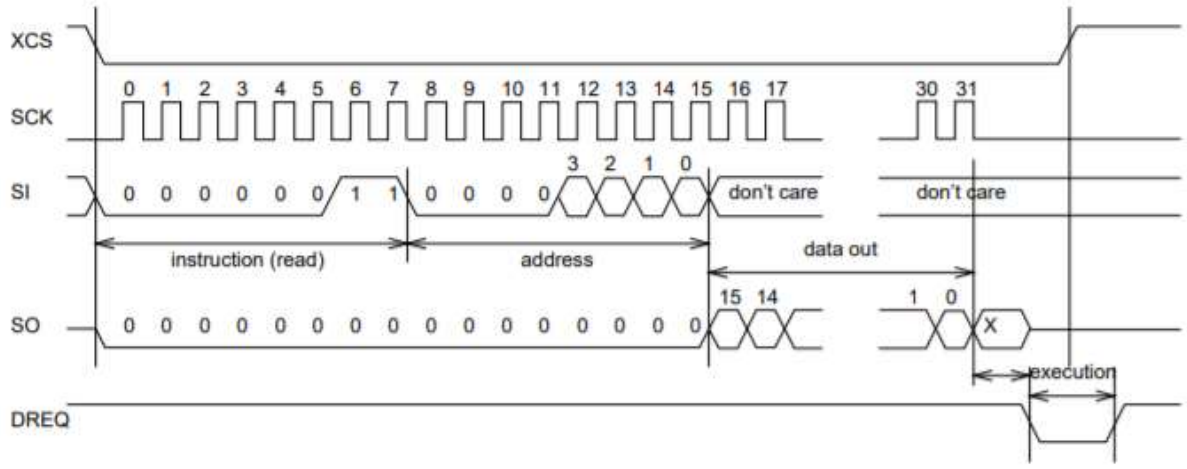



*Figure 18: Read operation: Taken from VS1053 datasheet. XCS refers to command select pin DREQ indicates whether chip is ready to receive next operation.*

**DATA WRITE**

It is important to note that the sequence of operations discussed here refers to the actual byte transfer of music data to the decode chip, which is where we faced most of our issues. XDCS pin is pulled low to initiate a data write. The datasheet explains there is a margin of 32 bytes which can be sent at once without toggling the chip select pins. The data write program checks the number of bytes to be transferred and toggles the XDCS (DATA chip select pin) between every 32 bytes of data transfer.

The DREQ pin plays a vital role here as it indicates whether the sent data has been processed by the decode and whether more data can be sent. Byte transfer can resume only when DREQ is pushed back to high again.

As there is no interface to read the actual data in the buffer of the decode chip, there is no actual value to verify whether data sent by the MCU is received appropriately by the decode chip.

### 2.3.2 MCU INTERFACING WITH SD CARD (SPI PROTOCOL)

SD card is accesses through SPI mode and by using specific command values as defined by the SD standard. The SD card was formatted with the FAT16 filesystem on a PC. FAT16 worked as the SD card's size was 1GB and FAT16 applies between 128 MB to 2GB.

For interfacing the SD card with the MSP 432 we used the FATS-SD DriverLIB example as a source and used the functions to mount the driver and have read access to the files. As the stated project didn't need write access to the files on the SD card (music files were loaded to the SD card through a PC).

The library example had firmware in the form of three layers: namely the underlying SPI functions, the FATFS library and interface DiskIO layer in order to use the SD card as an external disk with file handling functions.

The FATFS library is an implementation from [6] which has function implementations for FAT filesystem operations like directories, file access (opening, reading) and file management itself.

The example code run for verification was programmed to check if a file of a specific filename was present and if not create a file of the same name. Then, contents of that file were then copied to another file and displayed serially.

The specific file functions which were used in our project required by us to test the functionality of the VS1053 chip were:

f_mount- Mounts the disk volume,
f_open- To open a file from disk
f_read - To read data from file.

The return pointer values from these functions are checked for error statuses and data read from the SD flash memory is fed into the VS decode chip as data write operations. The written values were then checked again with the hex values of the audio files using a hex file editor and they were found to match.

### 2.3.3 MCU INTERFACING WITH WTV020 MODULE

The WTV020 module [3] is a simpler audio module which can directly access SD card files with a 2-pin command interface. One pin serves as the clock pin (generated by the connected MCU) and the other pin serves as a data line. The major source of firmware example we used was Sparkfun's own Arduino example code [7]for the module which we had to adapt for the MSP432.

The module specifically looks for filenames in the SD card and accesses these files to play them. The format of the filenames themselves "0000 .ad4", "0001.ad4" etc. Therefore, when a command is given to play song number '0', the module looks for file "0000.ad4" to play. The ad4 file format is also the only file format, which is supported by the module, forcing use to use a utility provided by Sparkfun to convert audio files to that format.

The send command routine is responsible for sending the command codes and is vital for any operation on the audio module. It is done by manually generating a pulse on the clock pin and

then bit banging the data line according to the data value to be sent. It has a reset pin which needs to be cleared and set with a specific delay in between in order to put the chip in a known state and a busy pin through which we can check if playback is currently running on the module, this pin is set as input on the MCU.

The module maps multiple pins to specific operations on the device but by going through the command operations available, we decided to restrict the number of pins in use and use only different commands sequentially to operate playback on the device. This allows us to save on the number of pins required even though we don't lose on any major functionality offered by the device. The operations offered by our driver includes: play/pause, change in volume and skipping songs.

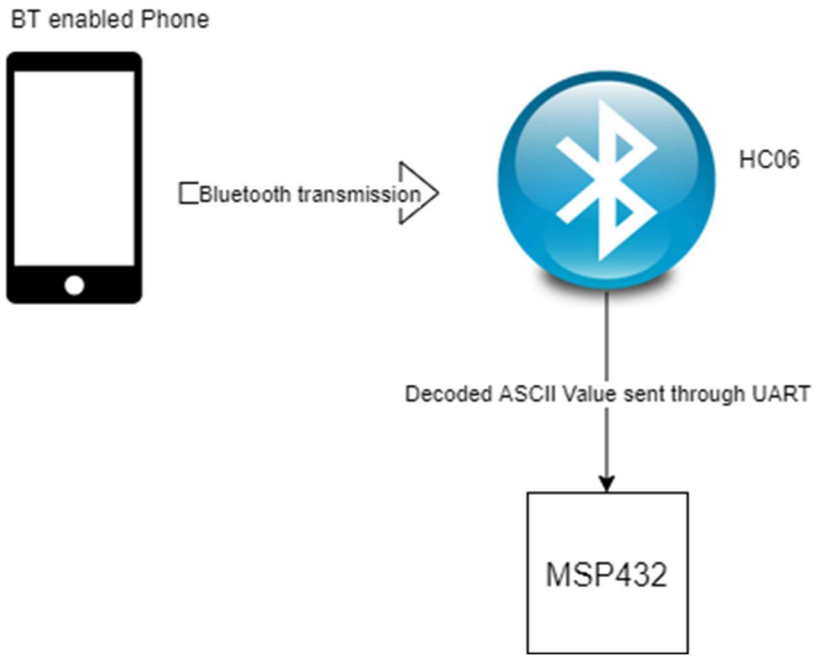
Though the module is easy to work with it is very sensitive with timing between the clock signals and the reset sequence. We used a timer block to deliver the exact amount of delay required and verified the timings by oscilloscope readings. Once the timings were close, we could get the module to function properly and allowed us to save time by not slowing down program functionality.

### 2.3.4 MCU INTERFACING WITH HC-06 BLUETOOTH (UART)

The HC-06 [4] is a Bluetooth module which can only act as a BT receiver and is interfaced with the MSP432 with a UART interface with 9600 baud rate. The module allows us to provide a Bluetooth interface to the audio player.

Initialisation settings (Using DriverLib):

```
const eUSCI_UART_ConfigV1 uartConfig0 =
{
 EUSCI_A_UART_CLOCKSOURCE_SMCLK,          // SMCLK Clock Source-3MHz
 19,                                     // BRDIV = 19
 8,                                       // UCxBRF = 2
 0,                                       // UCxBRS = 0
 EUSCI_A_UART_NO_PARITY,                  // No Parity
 EUSCI_A_UART_LSB_FIRST,                  // LSB First
 EUSCI_A_UART_ONE_STOP_BIT,               // One stop bit
 EUSCI_A_UART_MODE,                       // UART mode
 EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION,  // Oversampling
 EUSCI_A_UART_8_BIT_LEN                  // 8 bit data length
};
```




With specific ASCII values mapped to routines, the HC06 module allows playback on the audio module to be controlled by the command sent by the phone.

### 2.3.5 MCU INTERFACING WITH LCD OPTREX 20434

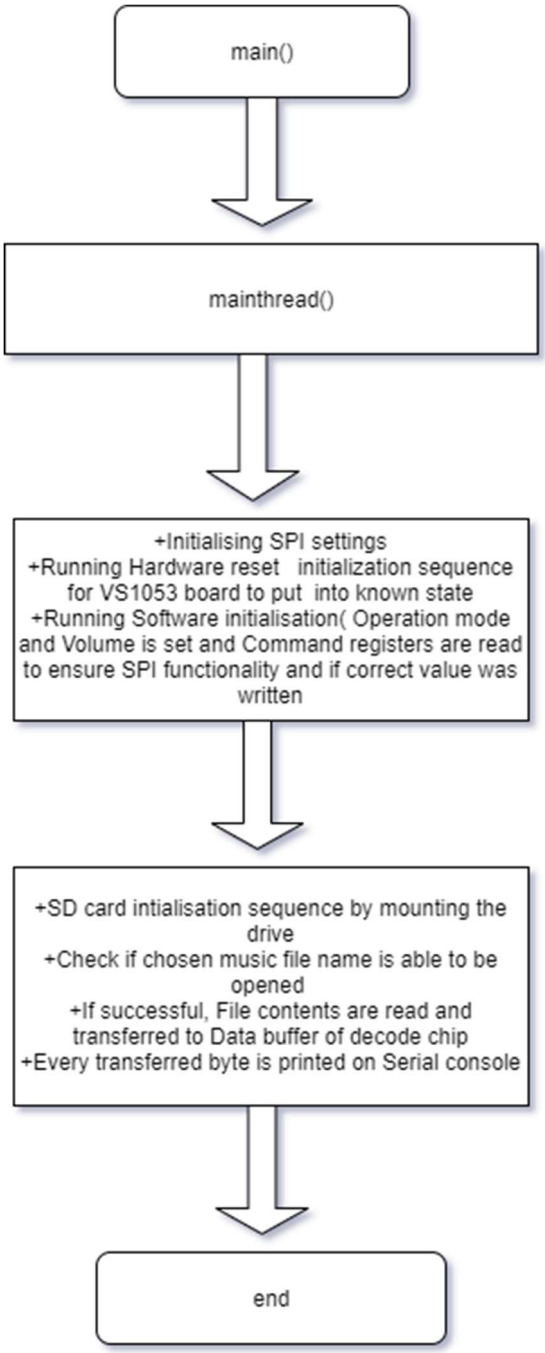
The LCD driver for Optrex 20344 LCD [8] unit was adapted from the work we had done in the course lab assignments. The difference between the two was that we couldn't use address mapping on the MSP432 and had to resort to set and clear the Enable, R/S (Register Select) and R/W using the GPIO pins of the MSp432.

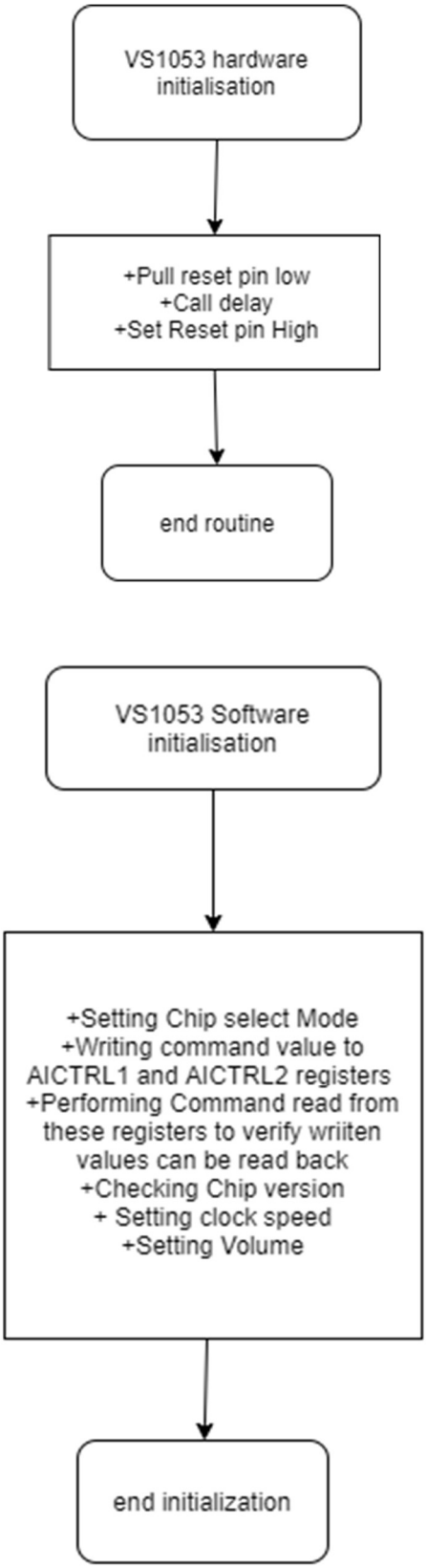
Apart from the above, the only other difference was to adjust the delay timings and running them longer due to the change in how the command pins of the LCD are operated.

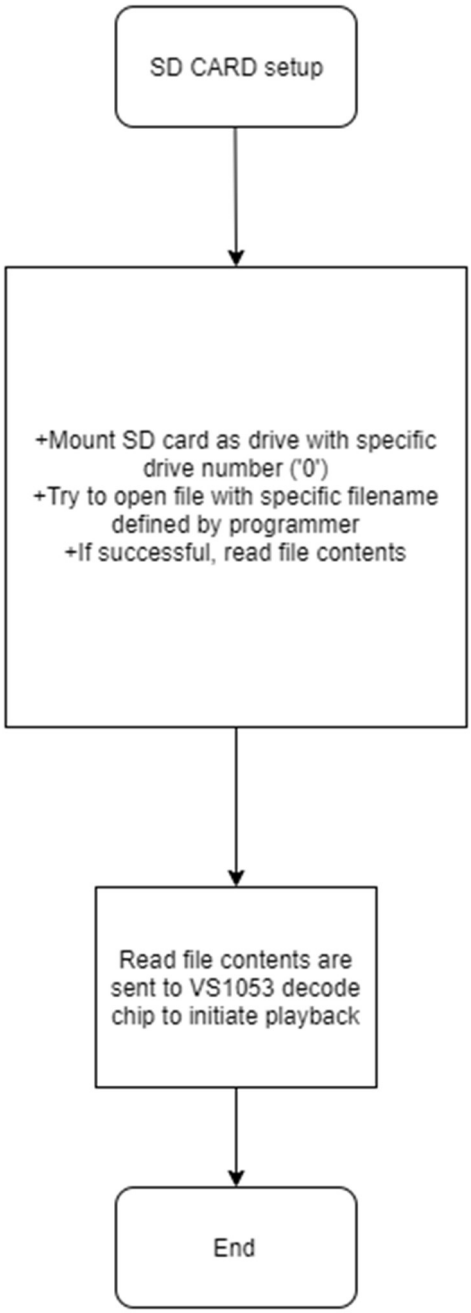
The LCD was programmed to reflect any playback status changes so all it required was a fixed character string to be displayed at appropriate times and to ensure that previous contents were cleared before every write.

## 2.4 SOFTWARE DESIGN

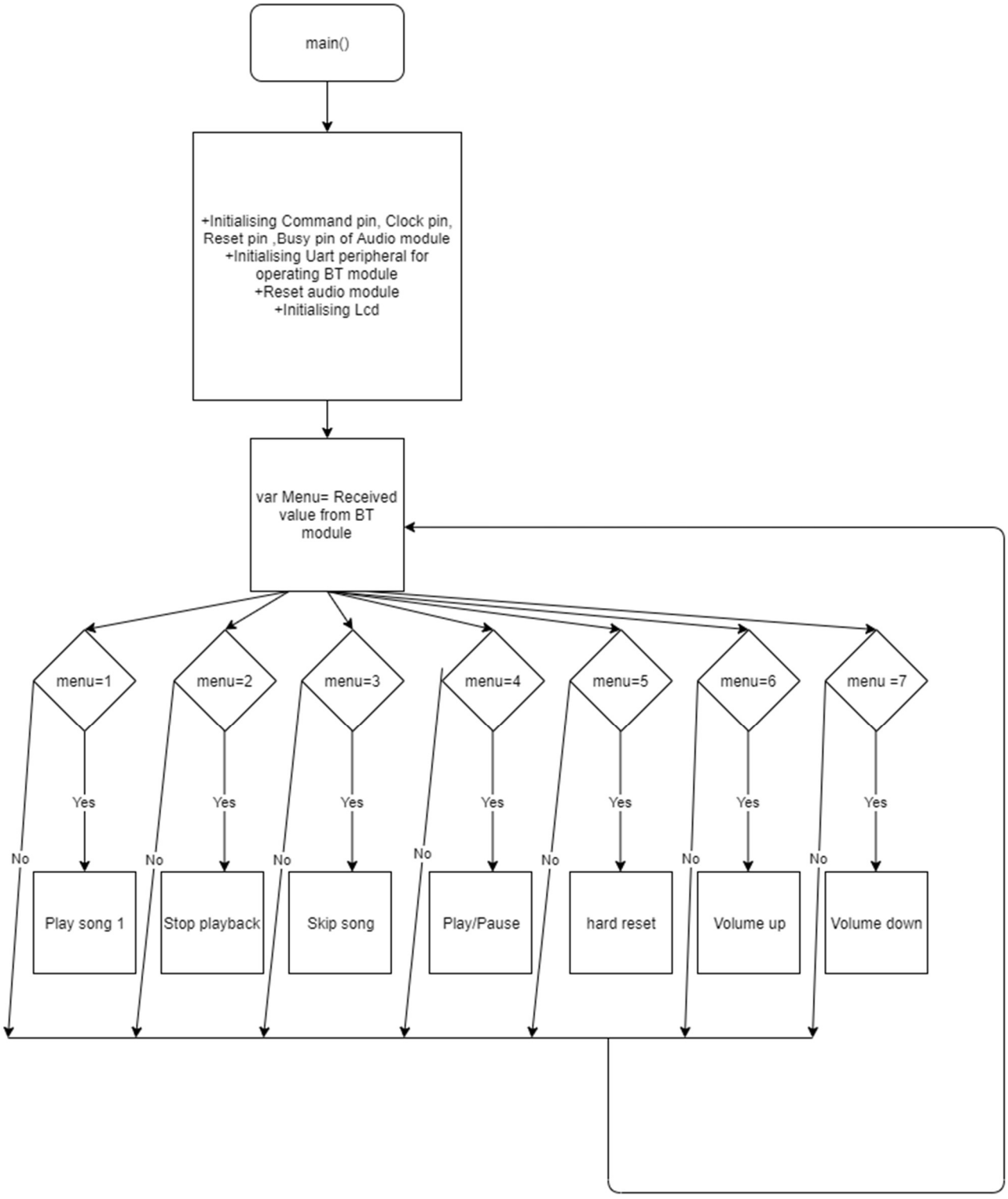
### VS1053 Routine




VS1053 hardware initialisation
+Pull reset pin low
+Call delay
+Set Reset pin High
end routine
VS1053 Software initialisation
+Setting Chip select Mode
+Writing command value to AICTRL1 and AICTRL2 registers
+Performing Command read from these registers to verify wriiten values can be read back
+Checking Chip version
+ Setting clock speed
+Setting Volume
end initialization


SD CARD setup
+Mount SD card as drive with specific drive number ('0')
+Try to open file with specific filename defined by programmer
+If successful, read file contents
Read file contents are sent to VS1053 decode chip to initiate playback
End

## MP3 Audio Module



# 3. ISSUES & ERROR ANALYSIS

While designing this project we faced issues at multiple stages, and we tried to adopt divergent approaches to approach a solution. The issues and the solution to those problems are interpreted below.

i. VS1053 Breakout Board Hardware Defect:

This is the major issue which we faced while designing the project using first approach. On interfacing the VS1053 with MSP-432 using SPI protocol, we were able to observe the hex values in the lookup table of WAV files stored in SD card being transferred to the VS1053 correctly. To verify this, we printed those values on the console before transferring them and compared them with the lookup table values. Also, the SD card was being shown as mounted and Output text file was getting generated from input text file. This helped us to investigate that it wasn't the issue from SD card interfacing side or the VS1053 interfacing side (using SPI) since data was getting retrieved correctly from the SD card and getting transferred to VS1053 decoder. So, we came to a conclusion that the board had issue on the audio jack side because of which, we couldn't listen to any audio through earphones.

Solution: Due to the issues in the audio jack side of breakout board which could had damaged some other components in the periphery, we decided to switch to WTV020SD MicroSD module as MP3 Decoder +

Takeaway from this issue: Thus, we tried to narrow down the error possibilities by validating the workability of the modules in the system and tried to resolve the issue.

ii. Design of Custom VS1053 Board Hardware constraints:

One of our goals at the beginning of the project was to design the VS1053 board on our own using DIP-48 SMT VS1053 chip and LQFP-48 Adaptor and designing subordinate circuitry. We started with the hand soldering process to solder the small DIP-48 chip and at the same time, we learnt about other techniques like using paste and desoldering in this process. However, the VS1053 board needed 12.288 Mhz clock which we failed to acquire in time. Because of this hardware constraint, we decided not to move further with

designing the whole board. However, we didn't want this constraint to hinder our learning process. So, we attempted and learnt about the soldering process in steps which can be seen in terms of flowchart below. The images captured while executing this process can be observed in figure 19 below.

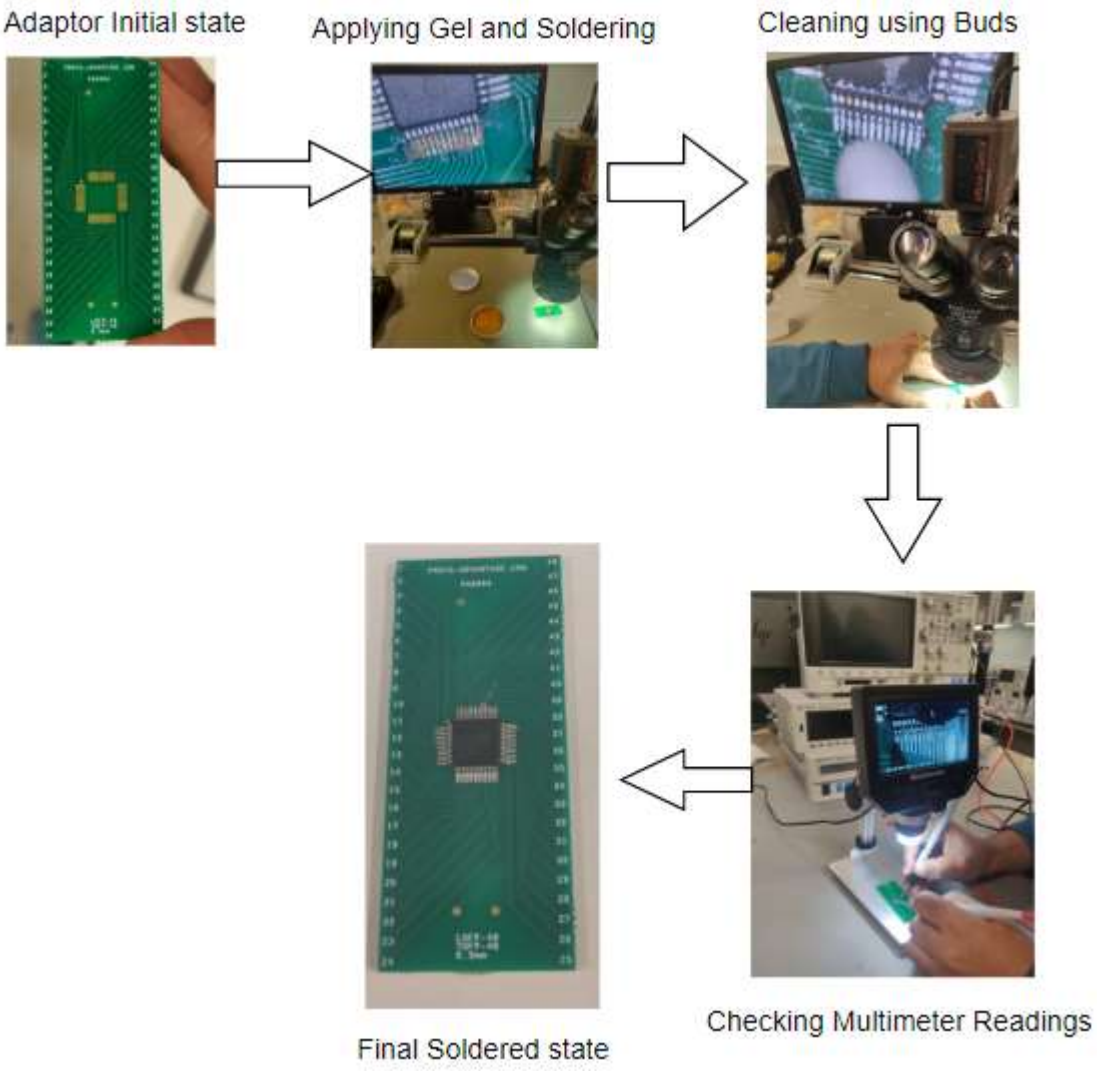



*Figure 19: Flow diagram of DIP- 48 chip soldering process*

Takeaway from the issue: While designing a prototype circuit on board, we should always look for the availability of all the components required to design the circuit. However, we could check if those values can be altered according to the availability of nearest value without varying the output value below acceptable error range. Also, the development of prototype should be

planned according to the timeline in which, we have to deliver the output and so time plays a crucial role in the development of embedded system and is a major parameter for tradeoff with other parameters such as size, cost and accuracy.

### iii. Noise Amplification in Audio Amplifier Circuitry

The Audio Amplifier designed in this project using LM386 audio amplifier IC provided amplified audio signals. The volume could be adjusted by varying the capacitance values between pin 1 and pin 8. Also, adding the capacitors in parallel to ground from the output at pin 5 helped to remove high frequency noise and smoothen the signals. However, we were getting the low frequency noise. To avoid this, we researched and found out that we needed to add a decoupling capacitor along with power source to remove low frequency noise. Also, the other challenge was that even though we had calculated the gain, we were skeptical whether the circuitry was working with desired gain specifications or not. These were the two issues which we had faced while designing the circuitry.

Solution: To solve the issue of low frequency noise, we added a 100uf decoupling capacitor at the output i.e. to pin 6 and parallel to GND. This helped us to attenuate the noise and improve the SNR (Signal to Noise Ratio).

To solve the second issue of checking the functionality of the circuit, we generated a sine wave using function generator and fed it as an input to the audio amplifier circuitry. And we observed the output on the oscilloscope to verify that we are getting amplified output in desired safe range.

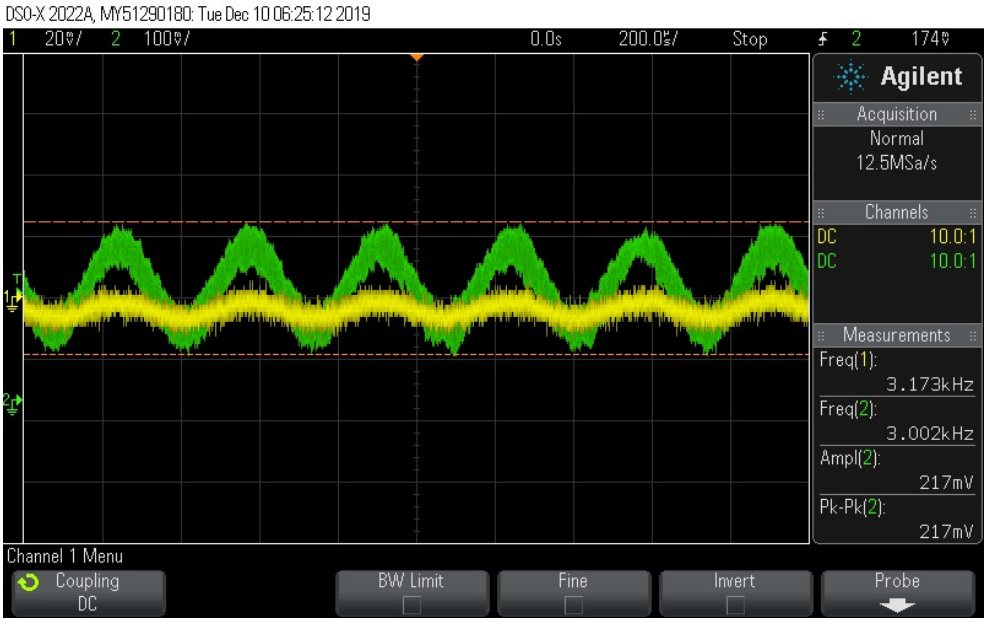



*Figure 20: Oscilloscope reading for amplified sine wave output*

Takeaway : Whenever we design a circuit related to power amplification or generating an output which strictly needs to be in specific range since the output outside that range could damage the other components then we should adapt disparate output verifying methods like using logic analyser or digital oscilloscope. This will help us to design a robust circuit and integrate it with the embedded system which is a superset of it.

### iv. SD card Shield and MicroSD Audio Module Workability testing:

To test whether the SD card shield was working properly and able to read the SD card contents, instead of using more expensive MSP-432 board to interface the shield, we decided to use a cheaper Arduino Board to interface and check the workability of the board. We loaded the SD card with a text file occupying some space in the card. Then, we executed sample code on Arduino IDE and verified the space left in the SD card by observing the output. We used Bone conduction transducer and MicroSD audio module interfaced with Arduino to test the working condition of the MicroSD Audio Module and the Bone Conduction Transducer. Figure 21 explains the Arduino Board interfacing with MicroSD module and Bone Conduction Module.

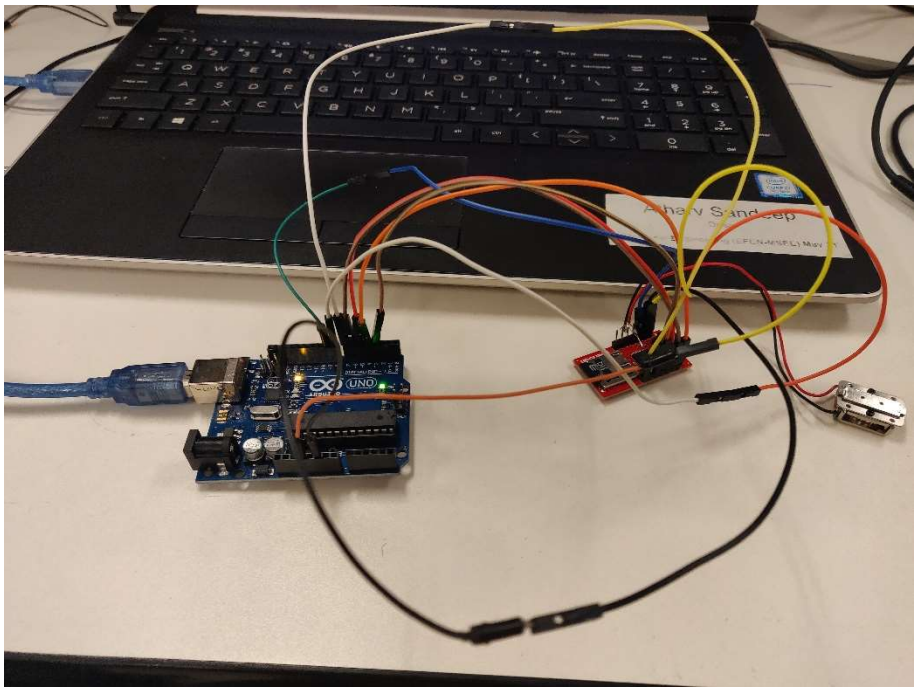


*Figure 21: Arduino Interfaced with MicroSD Audio Module and* Bone Conduction Transducer

Takeaway: Before interfacing the small components and modules with the expensive boards, we should use cheaper emulator or simulator to check workability of the small modules. This ensures that the whole embedded system doesn't get damaged on adding new small components while designing the prototype.

# 4. CONCLUSION

This project gave us deep insight about the various development stages in embedded system such as the motivation behind project, conceptualization as well as designing, testing and debugging and deployment of all the modules in the system. Writing Communication protocols like SPI, UART and doing bit banging helped us to improve my firmware writing skills.

We improved hardware design skills while designing Audio Amplifier circuitry, soldering DIP-48 chip and placing the HC-06 and MicroSD Audio Module on the perfboard.

Additionally, we learnt about when to hop to fallback plan by analyzing the time constraints, level of problems to be debugged and precision of the output to be expected by the user of that embedded system. In the end, listening to the song being played by our own MP3 player gave as a sense of elation and motivated us to work harder on further projects.

# 5. FUTURE DEVELOPMENT IDEAS

We feel that this project is open to a lot of improvements and additional functionalities. We have listed some of them below which we could had incorporated in our project or will in future.

i. Instead of using mobile phone app as a transmitter, we would want to use another HC-05 along with MSP-432 as master. This would help us to learn about AT commands and the process of binding the master to the slave.
ii. Additionally, instead of sending ASCII value to select the songs, we could had used a microphone at the transmitter MSP-432 side to allow voice recognition for selecting the songs.
iii. Instead of storing the songs in SD card, we can store the songs on the IoT cloud and access the songs from that cloud using MSP-432
iv. In the audio amplifier, if we could put a variable capacitor to vary the gain which will vary the volume. So, it can act as a volume button to our MP3 player.

# 6. References


1. VS1053 - http://www.vlsi.fi/fileadmin/datasheets/vs1053.pdf
2. SD CARD SHIELD - https://github.com/sparkfun/Shifting_microSD
3. WTV020SD: https://dlnmh9ip6v2uc.cloudfront.net/datasheets/Widgets/WTV020SD.pdf
4. HC-06 - https://www.fecegypt.com/uploads/dataSheet/1480849570_hc06.pdf
5. AUDIO AMPLIFIER - https://www.instructables.com/id/Tales-From-the-Chip-LM386-Audio-Amplifier/
6. FATSD library: http://elm-chan.org/fsw/ff/00index_e.html
7. MicroSD card module Arduino example :https://github.com/sparkfun/Audio-Sound_Breakout-WTV020SD/blob/V_1.0/firmware/Audio-Sound_Breakout/AudioModule.ino
8. LCD 20434 - http://ecee.colorado.edu/~mcclurel/hd44780u.pdf
9. MSP432 - http://www.ti.com/lit/ds/symlink/msp432p401r.pdf


# 7. Appendices

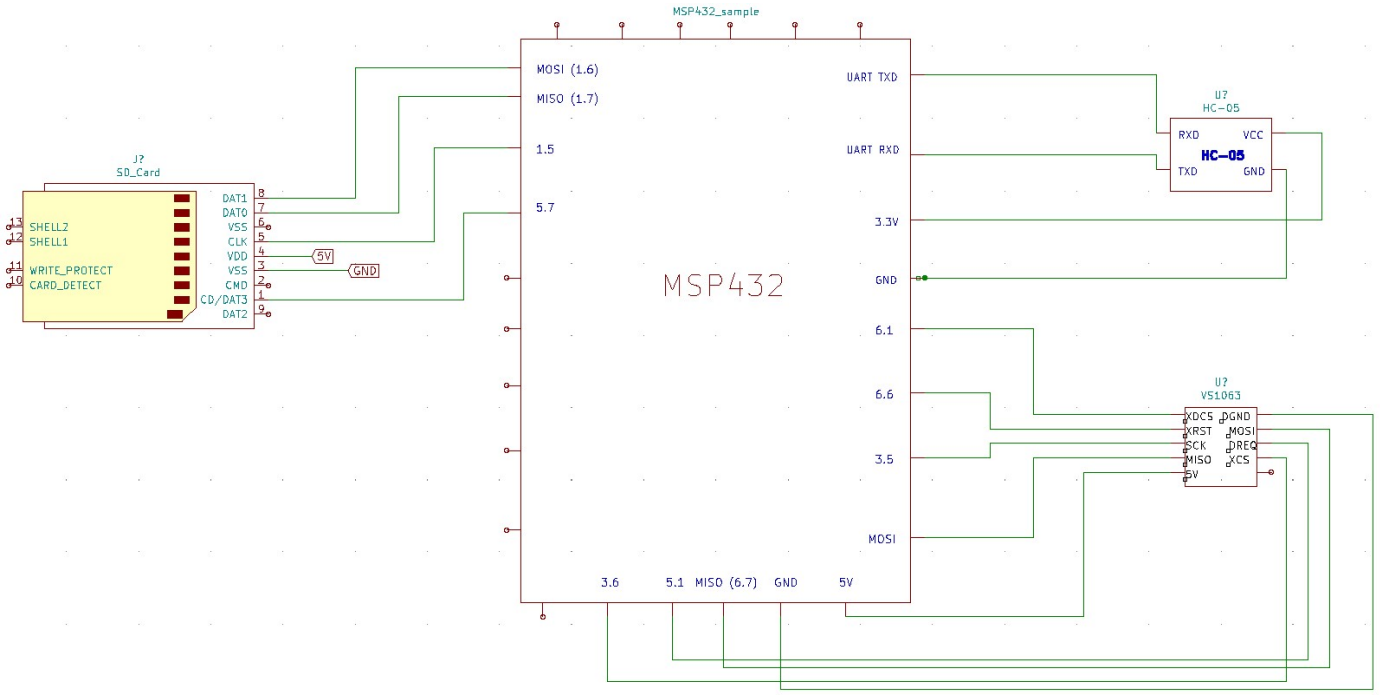
## 7.1 Bill of Materials

| COMPONENT | USE | COST |
|---|---|---|
| VS1053 DIP CHIP + ADAPTER | MAKING CUSTOM VS1053 BOARD | $ 13 |
| VS1053 BREAKOUT BOARD | AS A FALLBACK PLAN TO CUSTOM BOARD | $ 11 |
| HC-06 MODULE | TO ESTABLISH BLUETOOTH COMMUNICATION | $ 7 |
| LM386 + AUDIO AMPLIFIER CIRCUITRY | TO AMPLIFY THE LOW POWER AUDIO OUTPUT | $ 3 |
| SPARKFUN MICROSD AUDIO MODULE | FALLBACK PLAN TO THE FAULTY VS1053 BREAKOUT BOARD | $ 0 (Borrowed from senior) |
| 8 OHM SPEAKER | TO PLAY THE SONGS | $ 3 |
| SD CARD SHIELD | TO INTERFACE THE SD CARD WITH MSP432 | $ 5 |
| PERFBOARD | TO MOUNT THE HC-06 MODULE AND MICROSD MODULE ON IT | $ 1.5 |

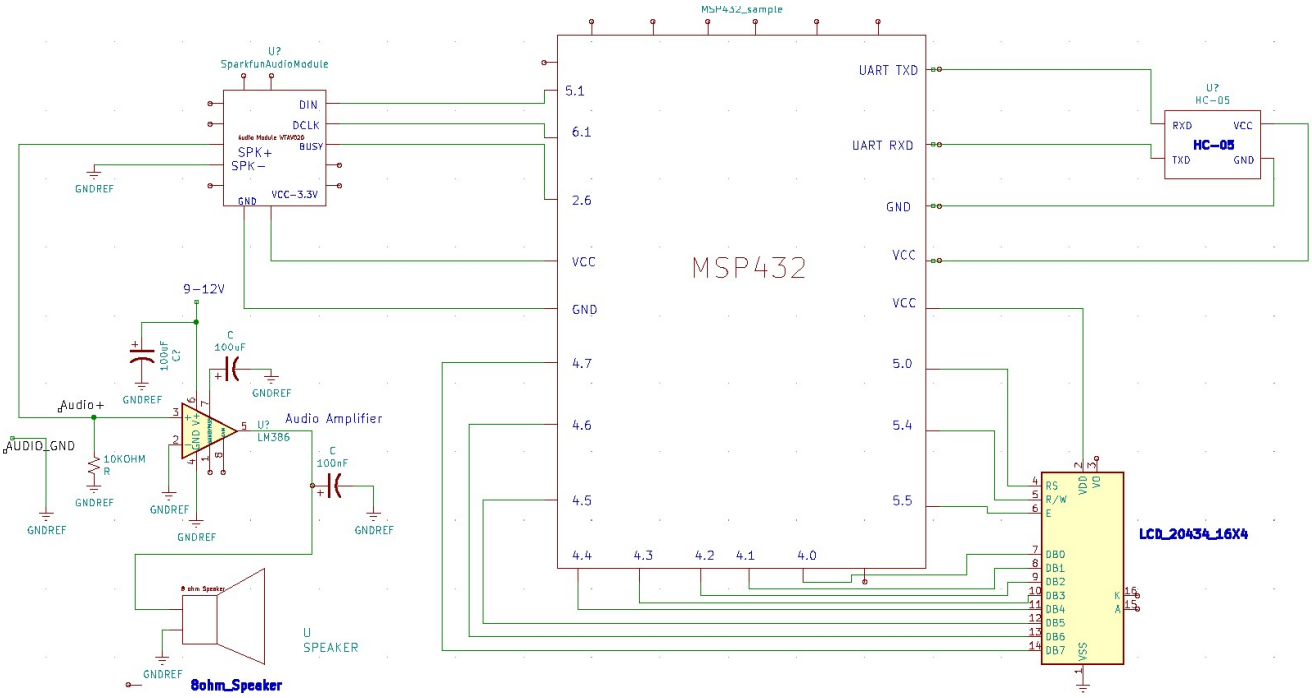
Therefore, the total budget of the project is $ 43.5.

## 7.2 SCHEMATIC

We have made the two schematics for first and second approaches. We have used KiCad Software to design the schematic.




*Schematic for first approach: VS1053 decoder with SD card adapter*




*Schematic for second approach*

## 7.3 Source Code

### VS1053 interface with SD Card

//// Authors: Sharan Arumugam and Atharv Desai

//MSP432 audio player

//SPI drivers for VS1053b breakout board. These functions are called by the player functions defined in a seperate file.

#include "ti/devices/msp432p4xx/inc/msp.h"

#include <stdint.h>

#include <stdio.h>

#include <stdbool.h>

void spi_init(); //sets up spi port settings. Initialises port pins and clock frequency

void spi_transmit(uint8_t datain);//responsible for transmitting data on spi bus

uint8_t spi_receive(); // responsible for receiving data on bus. returns 8 bit value

void EUSCIB2_IRQHandler(void);

```
//Code to handle VS1053 SPI bus - Authors: Sharan Arumugam and Atharv Desai
#include "spi_portb.h"
volatile uint8_t RXData = 0;
volatile bool bool_rx=0;


void spi_init()
{
        // Configure SPI for port 3 , other port for SD card
        EUSCI_B2->CTLW0 = 0;
        EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_SWRST; // Put eUSCI in reset
        EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_MST; //MSP432 in master mode
        EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_MSB; //MSB first mode
        EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_SYNC; //Synchronous
        EUSCI_B2->CTLW0 |= (EUSCI_B_CTLW0_CKPH); // Clock phase bit set high as per datasheet
requirements
        EUSCI_B2->CTLW0 |= EUSCI_B_CTLW0_SSEL__SMCLK|EUSCI_B_IE_RXIE0; //Clock source
selected as ACLK and receive interrupt enabled
        EUSCI_B2->BRW = 6; //This divides output frequency, by 6,


        //Configure GPIO pins to act as spi bus


        P3->SEL0 |= BIT5 | BIT6 | BIT7;
        P3->SEL1 &= ~(BIT5 | BIT6 | BIT7);
        //restart the state machine
        EUSCI_B2->CTLW0 &= ~(UCSWRST);
        //restart the state machine

                        __enable_irq();
   // Enable eUSCI_B0 interrupt in NVIC module
```

```
    NVIC->ISER[0] = 1 << ((EUSCIB2_IRQn) & 31);

}
void spi_transmit(uint8_t datain)
{  //P6->OUT |= BIT4;


    EUSCI_B2->TXBUF = datain;
    while(!(EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG));




    //P6->OUT &= ~BIT4;
}
uint8_t spi_receive()
{

while(!bool_rx)
    ;

bool_rx=false;
return RXData;
}

void EUSCIB2_IRQHandler(void)
{

    if (EUSCI_B2->IFG & EUSCI_B_IFG_RXIFG)
```

```
    {

    bool_rx=true;
    RXData = EUSCI_B2->RXBUF;
    }


  EUSCI_B2->IFG &= ~EUSCI_B_IFG_RXIFG;

}
```

////Code to handle VS1053 Command send and read functions and Data write instructions - Authors: Sharan Arumugam and Atharv Desai

```
#include "spi_portb.h"
//xcs pin5.1 xcs

#define setxcs P5->DIR |= BIT1;P5->OUT |= BIT1; //acts as command select pin
#define clearxcs P5->DIR |= BIT1;P5->OUT &= ~BIT1;
#define setxdcs P6->DIR |= BIT1;P6->OUT |= BIT1; //acts as data select pin
#define clearxdcs P6->DIR |= BIT1;P6->OUT &= ~BIT1;
#define CMDWRITE 0x2
#define CMDREAD 0x3
//dreq at p6.7
bool read_dreq(); //checks dreq pin to see if next operation can occur
void WriteSci( uint8_t address,uint16_t data); //command to write command code to chip
uint16_t ReadSci(uint8_t address); //reads value from cmd register
void WriteSdi(uint8_t *data, uint8_t bytes);//writes /sends data to chip buffer
#include "mp3_funcs.h"
```

```
////Code to handle VS1053 Command send and read functions and Data write instructions - Authors: Sharan Arumugam and Atharv Desai


bool read_dreq() //checks if next transaction can occue
{
   P6->DIR &=~BIT7;
   if(P6->IN & BIT7)
      return true;
   else
      return false;

}
void WriteSci( uint8_t address,uint16_t data)
{

   while(!(read_dreq()));

   clearxcs; //clearing xcs
   spi_transmit(CMDWRITE);

   spi_transmit(address);

   uint8_t byte =0;
   byte= (data)>>8;
   spi_transmit(byte);
   byte=data&0x00FF;


   spi_transmit(byte);
```

```
    int i=0;
    while(i!=100) //time to finish last transmission
        i++;
    setxcs; //setting xcs after transmission
    while(!(read_dreq())); //checking if dreq has been st to indicate operation is open


}

uint16_t ReadSci(uint8_t address)
{   while((P6->IN & BIT7)==0);
    uint16_t readvalue=0;
    clearxcs;
    spi_transmit(CMDREAD);
    spi_transmit(address);
    EUSCI_B2->TXBUF=0XFF;
    while(!(EUSCI_B2->IFG & EUSCI_B_IFG_TXIFG)){};
    EUSCI_B2->TXBUF=0XFF;
    while(!(EUSCI_B2->IFG & EUSCI_B_IFG_TXIFG)){};
    EUSCI_B2->IFG &=~ EUSCI_B_IFG_TXIFG;

    readvalue|=(uint16_t)spi_receive()<<8;


    readvalue|=spi_receive();
    printf("received val:%d\n",readvalue);
    int i=0;
    while(i!=50) //time to finish last transmission
```

```
        i++;
    setxcs;
    return readvalue;
}



void  WriteSdi(uint8_t *data, uint8_t bytes)
{ while((P6->IN & BIT7)==0);
    if(bytes<32)
    {
        //setxdcs;
        clearxdcs;
        int i;
        for(i=0;i<bytes;i++)
        {
            spi_transmit(*data);
            data++;
        }

        int f =0;
        while(f!=50) //time to finish last transmission
            f++;
        setxdcs;
    }
    else
    {
        int bytecount=0;
        setxdcs;
        int f =0;
```

```
            while(f!=50) //time to finish last transmission
               f++;
      clearxdcs;
      int x;
      for(x=0;x<bytes;x++,bytecount++)
      {
         if(bytecount==31)
         {
            setxdcs;
            clearxdcs;
            bytecount=0;
         }
         spi_transmit(*data);
         printf("transmitted data was %X\n",*data);
         data++;
      }
      int i=0;
      while(i!=500) //time to finish last transmission
         i++;
      setxdcs;


   }
   while(!(read_dreq()));//checking if dreq has been st to indicate operation is open
}
//Sharan Arumugam -MSP432 audio player
//Code derived from C code example given by manufacture in VS1053 APPLICATION NOTE: PLAYBACK AND RECORDING. CODE ONLY REQUIRED FOR MICROCNTROLLER PSPECIFIC SPI FUNCTIONS
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <ctype.h>
#define setxreset P6->DIR |= BIT6;P5->OUT |= BIT6;
#define clearxreset P6->DIR |= BIT6;P5->OUT &= ~BIT6;
#include "mp3_funcs.h"
#include "vs10xx_uc.h"

u_int32 ReadVS10xxMem32Counter(u_int16 addr);
u_int32 ReadVS10xxMem32(u_int16 addr) ;
u_int16 ReadVS10xxMem(u_int16 addr) ;
void WriteVS10xxMem(u_int16 addr, u_int16 data);
void WriteVS10xxMem32(u_int16 addr, u_int32 data);
static u_int16 LinToDB(unsigned short n);
void LoadPlugin(const u_int16 *d, u_int16 len);
void VS1053PlayFile(FILE *readFp);
void Set16(u_int8 *d, u_int16 n);
int VSTestInitHardware(void) ; //init function puts chip in known state by toggling the reset pin
int VSTestInitSoftware(void); //software init function to set up with proper values and  perform command read to verify if spi fucntions work properly
//int VSTestHandleFile(const char *fileName, int record);
/*

 VLSI Solution generic microcontroller example player / recorder for
 VS1053.

 v1.10 2016-05-09 HH  Modified quick sanity check registers
 v1.03 2012-12-11 HH  Recording command 'p' was VS1063 only -> removed
                      Added chip type recognition
 v1.02 2012-12-04 HH  Command '_' incorrectly printed VS1063-specific fields
```

```
 v1.01 2012-11-28 HH  Untabified

 v1.00 2012-11-27 HH  First release

// ////Code to handle intitalisation and initiate playback on VS1053b  by Sharan Arumugam and Atharv Desai

//Taken from code given with the application note


*/


#include "player1053.h"

#include "vs1053b-patches.plg"

#define u_int16 uint16_t


#define u_int8 uint8_t






const uint16_t chipNumber[16] = {

                        1001, 1011, 1011, 1003, 1053, 1033, 1063, 1103,

                        0, 0, 0, 0, 0, 0, 0, 0

   };



#define FILE_BUFFER_SIZE 512

#define SDI_MAX_TRANSFER_SIZE 32

#define SDI_END_FILL_BYTES_FLAC 12288

#define SDI_END_FILL_BYTES      2050

#define REC_BUFFER_SIZE 512
```

```
#define REPORT_INTERVAL 4096
#define REPORT_INTERVAL_MIDI 512



#define min(a,b) (((a)<(b))?(a):(b))



enum AudioFormat {
  afUnknown,
  afRiff,
  afOggVorbis,
  afMp1,
  afMp2,
  afMp3,
  afAacMp4,
  afAacAdts,
  afAacAdif,
  afFlac,
  afWma,
  afMidi,
} audioFormat = afUnknown;

const char *afName[] = {
  "unknown",
  "RIFF",
```

```
  "Ogg",
  "MP1",
  "MP2",
  "MP3",
  "AAC MP4",
  "AAC ADTS",
  "AAC ADIF",
  "FLAC",
  "WMA",
  "MIDI",
};


/*
  Read 32-bit increasing counter value from addr.
  Because the 32-bit value can change while reading it,
  read MSB's twice and decide which is the correct one.
*/
u_int32 ReadVS10xxMem32Counter(u_int16 addr) {
  u_int16 msbV1, lsb, msbV2;
  u_int32 res;

  WriteSci(SCI_WRAMADDR, addr+1);
  msbV1 = ReadSci(SCI_WRAM);
  WriteSci(SCI_WRAMADDR, addr);
  lsb = ReadSci(SCI_WRAM);
  msbV2 = ReadSci(SCI_WRAM);
  if (lsb < 0x8000U) {
    msbV1 = msbV2;
```

```
  }
  res = ((u_int32)msbV1 << 16) | lsb;

  return res;
}



/*
  Read 32-bit non-changing value from addr.
*/
u_int32 ReadVS10xxMem32(u_int16 addr) {
  u_int16 lsb;
  WriteSci(SCI_WRAMADDR, addr);
  lsb = ReadSci(SCI_WRAM);
  return lsb | ((u_int32)ReadSci(SCI_WRAM) << 16);
}



/*
  Read 16-bit value from addr.
*/
u_int16 ReadVS10xxMem(u_int16 addr) {
  WriteSci(SCI_WRAMADDR, addr);
  return ReadSci(SCI_WRAM);
}



/*
  Write 16-bit value to given VS10xx address
```

```
*/
void WriteVS10xxMem(u_int16 addr, u_int16 data) {
  WriteSci(SCI_WRAMADDR, addr);
  WriteSci(SCI_WRAM, data);
}

/*
  Write 32-bit value to given VS10xx address
*/
void WriteVS10xxMem32(u_int16 addr, u_int32 data) {
  WriteSci(SCI_WRAMADDR, addr);
  WriteSci(SCI_WRAM, (u_int16)data);
  WriteSci(SCI_WRAM, (u_int16)(data>>16));
}




static const u_int16 linToDBTab[5] = {36781, 41285, 46341, 52016, 58386};

/*
  Converts a linear 16-bit value between 0..65535 to decibels.
    Reference level: 32768 = 96dB (largest VS1053b number is 32767 = 95dB).
  Bugs:
    - For the input of 0, 0 dB is returned, because minus infinity cannot
      be represented with integers.
    - Assumes a ratio of 2 is 6 dB, when it actually is approx. 6.02 dB.
*/
static u_int16 LinToDB(unsigned short n) {
```

```
  int res = 96, i;

  if (!n)             /* No signal should return minus infinity */
    return 0;

  while (n < 32768U) {  /* Amplify weak signals */
    res -= 6;
    n <<= 1;
  }

  for (i=0; i<5; i++)   /* Find exact scale */
    if (n >= linToDBTab[i])
      res++;

  return res;
}




/*

  Loads a plugin.

  This is a slight modification of the LoadUserCode() example
  provided in many of VLSI Solution's program packages.

*/
void LoadPlugin(const u_int16 *d, u_int16 len) {
```

```
 int i = 0;


 while (i<len) {
  unsigned short addr, n, val;
  addr = d[i++];
  n = d[i++];
  if (n & 0x8000U) { /* RLE run, replicate n samples */
    n &= 0x7FFF;
    val = d[i++];
    while (n--) {
      WriteSci(addr, val);
    }
  } else {          /* Copy run, copy n samples */
    while (n--) {
      val = d[i++];
      WriteSci(addr, val);
    }
  }
 }
}
```

```
enum PlayerStates {
  psPlayback = 0,
  psUserRequestedCancel,
  psCancelSentToVS10xx,
  psStopped
} playerState;




/*

  This function plays back an audio file.

  It also contains a simple user interface, which requires the following
  funtions that you must provide:
  void SaveUIState(void);
  - saves the user interface state and sets the system up
  - may in many cases be implemented as an empty function
  void RestoreUIState(void);
  - Restores user interface state before exit
  - may in many cases be implemented as an empty function
  int GetUICommand(void);
  - Returns -1 for no operation
  - Returns -2 for cancel playback command
  - Returns any other for user input. For supported commands, see code.

*/
```

```
void VS1053PlayFile(FILE *readFp) {
  static u_int8 playBuf[FILE_BUFFER_SIZE];
  u_int32 bytesInBuffer;       // How many bytes in buffer left
  u_int32 pos=0;               // File position
  int endFillByte = 0;         // What byte value to send after file
  int endFillBytes = SDI_END_FILL_BYTES; // How many of those to send
  int playMode = ReadVS10xxMem(PAR_PLAY_MODE);
  long nextReportPos=0; // File pointer where to next collect/report
  int i;
#ifdef PLAYER_USER_INTERFACE
  static int earSpeaker = 0;    // 0 = off, other values strength
  int volLevel = ReadSci(SCI_VOL) & 0xFF; // Assume both channels at same level
  int c;
  static int rateTune = 0;      // Samplerate fine tuning in ppm
#endif /* PLAYER_USER_INTERFACE */

#ifdef PLAYER_USER_INTERFACE
  SaveUIState();
#endif /* PLAYER_USER_INTERFACE */

  playerState = psPlayback;            // Set state to normal playback

  WriteSci(SCI_DECODE_TIME, 0);        // Reset DECODE_TIME


  /* Main playback loop */

  while ((bytesInBuffer = fread(playBuf, 1, FILE_BUFFER_SIZE, readFp)) > 0 &&
         playerState != psStopped) {
```

```
u_int8 *bufP = playBuf;

while (bytesInBuffer && playerState != psStopped) {

 if (!(playMode & PAR_PLAY_MODE_PAUSE_ENA)) {
  int t = min(SDI_MAX_TRANSFER_SIZE, bytesInBuffer);

  // This is the heart of the algorithm: on the following line
  // actual audio data gets sent to VS10xx.
  WriteSdi(bufP, t);

  bufP += t;
  bytesInBuffer -= t;
  pos += t;
 }

 /* If the user has requested cancel, set VS10xx SM_CANCEL bit */
 if (playerState == psUserRequestedCancel) {
  unsigned short oldMode;
  playerState = psCancelSentToVS10xx;
  printf("\nSetting SM_CANCEL at file offset %ld\n", pos);
  oldMode = ReadSci(SCI_MODE);
  WriteSci(SCI_MODE, oldMode | SM_CANCEL);
 }

 /* If VS10xx SM_CANCEL bit has been set, see if it has gone
   through. If it is, it is time to stop playback. */
 if (playerState == psCancelSentToVS10xx) {
  unsigned short mode = ReadSci(SCI_MODE);
```

```
      if (!(mode & SM_CANCEL)) {
        printf("SM_CANCEL has cleared at file offset %ld\n", pos);
        playerState = psStopped;
      }
    }


    /* If playback is going on as normal, see if we need to collect and
       possibly report */
    if (playerState == psPlayback && pos >= nextReportPos) {
#ifdef REPORT_ON_SCREEN
      u_int16 sampleRate;
      u_int32 byteRate;
      u_int16 h1 = ReadSci(SCI_HDAT1);
#endif

      nextReportPos += (audioFormat == afMidi || audioFormat == afUnknown) ?
        REPORT_INTERVAL_MIDI : REPORT_INTERVAL;
      /* It is important to collect endFillByte while still in normal
         playback. If we need to later cancel playback or run into any
         trouble with e.g. a broken file, we need to be able to repeatedly
         send this byte until the decoder has been able to exit. */
      endFillByte = ReadVS10xxMem(PAR_END_FILL_BYTE);

#ifdef REPORT_ON_SCREEN
      if (h1 == 0x7665) {
        audioFormat = afRiff;
        endFillBytes = SDI_END_FILL_BYTES;
      } else if (h1 == 0x4154) {
```

```
  audioFormat = afAacAdts;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if (h1 == 0x4144) {
  audioFormat = afAacAdif;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if (h1 == 0x574d) {
  audioFormat = afWma;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if (h1 == 0x4f67) {
  audioFormat = afOggVorbis;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if (h1 == 0x664c) {
  audioFormat = afFlac;
  endFillBytes = SDI_END_FILL_BYTES_FLAC;
 } else if (h1 == 0x4d34) {
  audioFormat = afAacMp4;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if (h1 == 0x4d54) {
  audioFormat = afMidi;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if ((h1 & 0xffe6) == 0xffe2) {
  audioFormat = afMp3;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if ((h1 & 0xffe6) == 0xffe4) {
  audioFormat = afMp2;
  endFillBytes = SDI_END_FILL_BYTES;
 } else if ((h1 & 0xffe6) == 0xffe6) {
  audioFormat = afMp1;
  endFillBytes = SDI_END_FILL_BYTES;
```

```
      } else {
        audioFormat = afUnknown;
        endFillBytes = SDI_END_FILL_BYTES_FLAC;
      }

      sampleRate = ReadSci(SCI_AUDATA);
      byteRate = ReadVS10xxMem(PAR_BYTERATE);
      /* FLAC:   byteRate = bitRate / 32
         Others: byteRate = bitRate /  8
         Here we compensate for that difference. */
      if (audioFormat == afFlac)
        byteRate *= 4;

      printf("\r%ldKiB "
           "%1ds %1.1f"
           "kb/s %dHz %s %s"
           " %04x   ",
           pos/1024,
           ReadSci(SCI_DECODE_TIME),
           byteRate * (8.0/1000.0),
           sampleRate & 0xFFFE, (sampleRate & 1) ? "stereo" : "mono",
           afName[audioFormat], h1
           );

      fflush(stdout);
#endif /* REPORT_ON_SCREEN */
    }
  } /* if (playerState == psPlayback && pos >= nextReportPos) */
```

```
  /* User interface. This can of course be completely removed and
     basic playback would still work. */

#ifdef PLAYER_USER_INTERFACE
  /* GetUICommand should return -1 for no command and -2 for CTRL-C */
  c = GetUICommand();
  switch (c) {

    /* Volume adjustment */
  case '-':
    if (volLevel < 255) {
      volLevel++;
      WriteSci(SCI_VOL, volLevel*0x101);
    }
    break;
  case '+':
    if (volLevel) {
      volLevel--;
      WriteSci(SCI_VOL, volLevel*0x101);
    }
    break;

    /* Show some interesting registers */
  case '_':
    {
      u_int32 mSec = ReadVS10xxMem32Counter(PAR_POSITION_MSEC);
      printf("\nvol %1.1fdB, MODE %04x, ST %04x, "
```

```
            "HDAT1 %04x HDAT0 %04x\n",
            -0.5*volLevel,
            ReadSci(SCI_MODE),
            ReadSci(SCI_STATUS),
            ReadSci(SCI_HDAT1),
            ReadSci(SCI_HDAT0));
      printf("  sampleCounter %lu, ",
            ReadVS10xxMem32Counter(0x1800));
      if (mSec != 0xFFFFFFFFU) {
        printf("positionMSec %lu, ", mSec);
      }
      printf("config1 0x%04x", ReadVS10xxMem(PAR_CONFIG1));
      printf("\n");
    }
    break;

    /* Adjust play speed between 1x - 4x */
  case '1':
  case '2':
  case '3':
  case '4':
    /* FF speed */
    printf("\nSet playspeed to %dX\n", c-'0');
    WriteVS10xxMem(PAR_PLAY_SPEED, c-'0');
    break;

    /* Ask player nicely to stop playing the song. */
  case 'q':
    if (playerState == psPlayback)
```

```
      playerState = psUserRequestedCancel;
    break;

    /* Forceful and ugly exit. For debug uses only. */
  case 'Q':
    RestoreUIState();
    printf("\n");
    exit(EXIT_SUCCESS);
    break;

    /* EarSpeaker spatial processing adjustment. */
  case 'e':
    earSpeaker = (earSpeaker+1) & 3;
    {
      u_int16 t = ReadSci(SCI_MODE) & ~(SM_EARSPEAKER_LO|SM_EARSPEAKER_HI);
      if (earSpeaker & 1)
        t |= SM_EARSPEAKER_LO;
      if (earSpeaker & 2)
        t |= SM_EARSPEAKER_HI;
      WriteSci(SCI_MODE, t);
    }
    printf("\nSet earspeaker to %d\n", earSpeaker);
    break;

    /* Toggle mono mode. Implemented in the VS1053b Patches package */
  case 'm':
    playMode ^= PAR_PLAY_MODE_MONO_ENA;
    printf("\nMono mode %s\n",
           (playMode & PAR_PLAY_MODE_MONO_ENA) ? "on" : "off");
```

```
  WriteVS10xxMem(PAR_PLAY_MODE, playMode);
  break;

  /* Toggle differential mode */
 case 'd':
  {
   u_int16 t = ReadSci(SCI_MODE) ^ SM_DIFF;
   printf("\nDifferential mode %s\n", (t & SM_DIFF) ? "on" : "off");
   WriteSci(SCI_MODE, t);
  }
  break;

  /* Adjust playback samplerate finetuning, this function comes from
     the VS1053b Patches package. Note that the scale is different
     in VS1053b and VS1063a! */
 case 'r':
  if (rateTune >= 0) {
   rateTune = (rateTune*0.95);
  } else {
   rateTune = (rateTune*1.05);
  }
  rateTune -= 1;
 if (rateTune < -160000)
   rateTune = -160000;
  WriteVS10xxMem(0x5b1c, 0);              /* From VS105b Patches doc */
  WriteSci(SCI_AUDATA, ReadSci(SCI_AUDATA)); /* From VS105b Patches doc */
  WriteVS10xxMem32(PAR_RATE_TUNE, rateTune);
  printf("\nrateTune %d ppm*2\n", rateTune);
  break;
```

```
  case 'R':
   if (rateTune <= 0) {
    rateTune = (rateTune*0.95);
   } else {
    rateTune = (rateTune*1.05);
   }
   rateTune += 1;
   if (rateTune > 160000)
    rateTune = 160000;
   WriteVS10xxMem32(PAR_RATE_TUNE, rateTune);
   WriteVS10xxMem(0x5b1c, 0);              /* From VS105b Patches doc */
   WriteSci(SCI_AUDATA, ReadSci(SCI_AUDATA)); /* From VS105b Patches doc */
   printf("\nrateTune %d ppm*2\n", rateTune);
   break;
  case '/':
   rateTune = 0;
   WriteVS10xxMem(SCI_WRAMADDR, 0x5b1c);    /* From VS105b Patches doc */
   WriteVS10xxMem(0x5b1c, 0);              /* From VS105b Patches doc */
   WriteVS10xxMem32(PAR_RATE_TUNE, rateTune);
   printf("\nrateTune off\n");
   break;

   /* Show help */
  case '?':
   printf("\nInteractive VS1053 file player keys:\n"
        "1-4\tSet playback speed\n"
        "- +\tVolume down / up\n"
        "_\tShow current settings\n"
        "q Q\tQuit current song / program\n"
```

```
            "e\tSet earspeaker\n"
            "r R\tR rateTune down / up\n"
            "/\tRateTune off\n"
            "m\tToggle Mono\n"
            "d\tToggle Differential\n"
            );
      break;

      /* Unknown commands or no command at all */
    default:
      if (c < -1) {
        printf("Ctrl-C, aborting\n");
        fflush(stdout);
        RestoreUIState();
        exit(EXIT_FAILURE);
      }
      if (c >= 0) {
        printf("\nUnknown char '%c' (%d)\n", isprint(c) ? c : '.', c);
      }
      break;
    } /* switch (c) */
#endif /* PLAYER_USER_INTERFACE */
  } /* while ((bytesInBuffer = fread(...)) > 0 && playerState != psStopped) */



#ifdef PLAYER_USER_INTERFACE
  RestoreUIState();
#endif /* PLAYER_USER_INTERFACE */
```

```
 printf("\nSending %d footer %d's... ", endFillBytes, endFillByte);
 fflush(stdout);

 /* Earlier we collected endFillByte. Now, just in case the file was
    broken, or if a cancel playback command has been given, write
    lots of endFillBytes. */
 memset(playBuf, endFillByte, sizeof(playBuf));
 for (i=0; i<endFillBytes; i+=SDI_MAX_TRANSFER_SIZE) {
  WriteSdi(playBuf, SDI_MAX_TRANSFER_SIZE);
 }

 /* If the file actually ended, and playback cancellation was not
    done earlier, do it now. */
 if (playerState == psPlayback) {
  unsigned short oldMode = ReadSci(SCI_MODE);
  WriteSci(SCI_MODE, oldMode | SM_CANCEL);
  printf("ok. Setting SM_CANCEL, waiting... ");
  fflush(stdout);
  while (ReadSci(SCI_MODE) & SM_CANCEL)
    WriteSdi(playBuf, 2);
 }

 /* That's it. Now we've played the file as we should, and left VS10xx
    in a stable state. It is now safe to call this function again for
    the next song, and again, and again... */
 printf("ok\n");
}
```

```
u_int8 adpcmHeader[60] = {
 'R', 'I', 'F', 'F',
 0xFF, 0xFF, 0xFF, 0xFF,
 'W', 'A', 'V', 'E',
 'f', 'm', 't', ' ',
 0x14, 0, 0, 0,         /* 20 */
 0x11, 0,               /* IMA ADPCM */
 0x1, 0,                /* chan */
 0x0, 0x0, 0x0, 0x0,    /* sampleRate */
 0x0, 0x0, 0x0, 0x0,    /* byteRate */
 0, 1,                  /* blockAlign */
 4, 0,                  /* bitsPerSample */
 2, 0,                  /* byteExtraData */
 0xf9, 0x1,             /* samplesPerBlock = 505 */
 'f', 'a', 'c', 't',    /* subChunk2Id */
 0x4, 0, 0, 0,          /* subChunk2Size */
 0xFF, 0xFF, 0xFF, 0xFF, /* numOfSamples */
 'd', 'a', 't', 'a',
 0xFF, 0xFF, 0xFF, 0xFF
};
```

```
u_int8 pcmHeader[44] = {
 'R', 'I', 'F', 'F',
 0xFF, 0xFF, 0xFF, 0xFF,
 'W', 'A', 'V', 'E',
 'f', 'm', 't', ' ',
 0x10, 0, 0, 0,         /* 16 */
 0x1, 0,               /* PCM */
 0x1, 0,               /* chan */
 0x0, 0x0, 0x0, 0x0,     /* sampleRate */
 0x0, 0x0, 0x0, 0x0,     /* byteRate */
 2, 0,                 /* blockAlign */
 0x10, 0,              /* bitsPerSample */
 'd', 'a', 't', 'a',
 0xFF, 0xFF, 0xFF, 0xFF
};

void Set32(u_int8 *d, u_int32 n) {
 int i;
 for (i=0; i<4; i++) {
   *d++ = (u_int8)n;
   n >>= 8;
 }
}

void Set16(u_int8 *d, u_int16 n) {
 int i;
 for (i=0; i<2; i++) {
   *d++ = (u_int8)n;
   n >>= 8;
```

```
  }
}
```

```
/*

  Hardware Initialization for VS1053.


*/
int VSTestInitHardware(void) {
   //xreset connected to p6.6

clearxreset;
int i=0;
while(i!=1000)
    i++;
setxreset;
  return 0;
}
```

```
/*
  Software Initialization for VS1053.
```

```
  Note that you need to check whether SM_SDISHARE should be set in
  your application or not.

*/

int VSTestInitSoftware(void)
{
  u_int16 ssVer;

  /* Start initialization with a dummy read, which makes sure our
     microcontoller chips selects and everything are where they
     are supposed to be and that VS10xx's SCI bus is in a known state. */
  ReadSci(SCI_MODE);

  /* First real operation is a software reset. After the software
     reset we know what the status of the IC is. You need, depending
     on your application, either set or not set SM_SDISHARE. See the
     Datasheet for details. */
  WriteSci(SCI_MODE, SM_SDINEW|SM_SDISHARE|SM_TESTS|SM_RESET);

  /* A quick sanity check: write to two registers, then test if we
     get the same results. Note that if you use a too high SPI
     speed, the MSB is the most likely to fail when read again. */
  WriteSci(SCI_AICTRL1, 0xABAD);
  WriteSci(SCI_AICTRL2, 0x7E57);
  if (ReadSci(SCI_AICTRL1) != 0xABAD || ReadSci(SCI_AICTRL2) != 0x7E57) { //performs a read of
command registers to verfy if written command values can be read successfully
    printf("There is something wrong with VS10xx SCI registers\n");
    return 1;
```

```
 }
 WriteSci(SCI_AICTRL1, 0);
 WriteSci(SCI_AICTRL2, 0);

 /* Check VS10xx type */
 ssVer = ((ReadSci(SCI_STATUS) >> 4) & 15);
 if (chipNumber[ssVer]) {
  printf("Chip is VS%d\n", chipNumber[ssVer]);
  if (chipNumber[ssVer] != 1053) {
   printf("Incorrect chip\n");
   return 1;
  }
 } else {
  printf("Unknown VS10xx SCI_MODE field SS_VER = %d\n", ssVer);
  return 1;
 }

 /* Set the clock. Until this point we need to run SPI slow so that
   we do not exceed the maximum speeds mentioned in
   Chapter SPI Timing Diagram in the Datasheet. */
 WriteSci(SCI_CLOCKF,
       HZ_TO_SC_FREQ(12288000) | SC_MULT_53_35X | SC_ADD_53_10X);


 /* Now when we have upped the VS10xx clock speed, the microcontroller
   SPI bus can run faster. Do that before you start playing or
   recording files. */

 /* Set up other parameters. */
```

```
  WriteVS10xxMem(PAR_CONFIG1, PAR_CONFIG1_AAC_SBR_SELECTIVE_UPSAMPLE);

  /* Set volume level at -6 dB of maximum */
  WriteSci(SCI_VOL, 0x0c0c);

  /* Now it's time to load the proper patch set. */
  LoadPlugin(plugin, sizeof(plugin)/sizeof(plugin[0]));

  /* We're ready to go. */
  return 0;
}




/*
  Main function that activates either playback or recording.
*/
int VSTestHandleFile(const char *fileName, int record) {
  if (!record) {
    FILE *fp = fopen(fileName, "rb");
    printf("Play file %s\n", fileName);
    if (fp) {
      VS1053PlayFile(fp);
    } else {
      printf("Failed opening %s for reading\n", fileName);
      return -1;
    }
```

```
 }


 return 0;

}
/*
 * Copyright (c) 2017-2019, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/*
 *  ======== fatsd.c ========
 */
/*
 Defines main thread which is called by main function
 Includes player1053.h functions in order to initiate playback
 Edited example file by Sharan Arumugam and Atharv Desai
 */
#include <file.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <time.h>
#include "mp3_funcs.h"
#include <third_party/fatfs/ffcio.h>
#include "player1053.h"
#include "spi_portb.h"
#include <ti/display/Display.h>
#include <ti/drivers/GPIO.h>
#include <ti/drivers/SDFatFS.h>

/* Driver configuration */
#include "ti_drivers_config.h"

/* Buffer size used for the file copy process */
#ifndef CPY_BUFF_SIZE
#define CPY_BUFF_SIZE       2048
#endif

/* String conversion macro */
#define STR_(n)             #n
#define STR(n)              STR_(n)

/* Drive number used for FatFs */
#define DRIVE_NUM           0

const char inputfile[] = "fat:"STR(DRIVE_NUM)":1.mp3";
const char outputfile[] = "fat:"STR(DRIVE_NUM)":out.mp3";

const char textarray[] = \
        "*********************************************************************\n"
        "0         1         2         3         4         5         6         7\n"
        "01234567890123456789012345678901234567890123456789012345678901234567890\n"
        "This is some text to be inserted into the inputfile if there isn't\n"
        "already an existing file located on the media.\n"
        "If an inputfile already exists, or if the file was already once\n"
        "generated, then the inputfile will NOT be modified.\n"
        "*********************************************************************\n";

static Display_Handle display;

/* File name prefix for this filesystem for use with TI C RTS */
char fatfsPrefix[] = "fat";

unsigned char cpy_buff[CPY_BUFF_SIZE + 1];


 */
void *mainThread(void *arg0)
{
    SDFatFS_Handle sdfatfsHandle;

    /* Variables for the CIO functions */
    FILE *src, *dst;
```

```
    /* Call driver init functions */
    GPIO_init();
    Display_init();
    SDFatFS_init();
    spi_init();
    /* Configure the LED pin */
    GPIO_setConfig(CONFIG_GPIO_LED_0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);

    /* add_device() should be called once and is used for all media types */
    add_device(fatfsPrefix, _MSA, ffcio_open, ffcio_close, ffcio_read,
               ffcio_write, ffcio_lseek, ffcio_unlink, ffcio_rename);

    /* Open the display for output */
    display = Display_open(Display_Type_UART, NULL);

    GPIO_write(CONFIG_GPIO_LED_0, CONFIG_GPIO_LED_ON);
    VSTestInitHardware();
    VSTestInitSoftware();

    Display_printf(display, 0, 0, "Starting the fatsd example\n");
    Display_printf(display, 0, 0,
                   "This example requires a FAT filesystem on the SD card.\n");
    Display_printf(display, 0, 0,
                   "You will get errors if your SD card is not formatted with a
filesystem.\n");

    /* Mount and register the SD Card */
    sdfatfsHandle = SDFatFS_open(CONFIG_SDFatFS_0, DRIVE_NUM);
    if (sdfatfsHandle == NULL) {
        Display_printf(display, 0, 0, "Error starting the SD card\n");
        while (1);
    }
    else {
        printf( "Drive %u is mounted\n", DRIVE_NUM);
    }

    /* Try to open the source file */
    src = fopen("sample.wav", "r"); //music file called smple.wav on sd card
    if(src)
        printf("file opened\n");

    WriteSci(SCI_VOL, 0x2020);
    VS1053PlayFile(src);



SDFatFS_close(sdfatfsHandle);
Display_printf(display, 0, 0, "Drive %u unmounted\n", DRIVE_NUM);

return (NULL);
}

/*
 *  ======== fatfs_getFatTime ========
 */
```

```
int32_t fatfs_getFatTime(void)
{
    /*
     *  FatFs uses this API to get the current time in FatTime format.  User's
     *  must implement this function based on their system's timekeeping
     *  mechanism.  See FatFs documentation for details on FatTime format.
     */
    /* Jan 1 2017 00:00:00 */
    return (0x4A210000);
}
/*
Running edited mainthread from fatsd_nortos.c
Edits made by Sharan Arumugam and Atharv Desai to FATSD DRiverlib example provided by
TI resource explorer
/*
 *  ======== main_nortos.c ========
 */
#include <stdint.h>
#include <stddef.h>

#include <NoRTOS.h>

/* Driver configuration */
#include <ti/drivers/Board.h>

extern void *mainThread(void *arg0);

/*//
 *  ======== main ========
 */
int main(void)
{
    /* Call driver init functions */
    Board_init();

    /* Start NoRTOS */
    NoRTOS_start();

    /* Call mainThread function */
    mainThread(NULL);

    return 0;
    while (1) {}
}
```

## MicroSD audio module, BT module

//ADAPTED FROM SPARFUNS ADRUINO FIRMWARE EXAMPLE:https://github.com/sparkfun/Audio-Sound_Breakout-WTV020SD/blob/V_1.0/firmware/Audio-Sound_Breakout/AudioModule.ino

//Code written for msp432 by Sharan Arumugam and Atharv Desai

```
#include <stdbool.h>
#include <stdint.h>
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>

#define setcommandpin P5->OUT |= BIT1
#define clearcommandpin P5->OUT &= ~BIT1
#define setclkpin P6->OUT |= BIT1
#define clearclkpin P6->OUT &= ~BIT1
#define setrstpin P5->OUT |= BIT6
#define clearrstpin P5->OUT &= ~BIT6
#define play_pause 0xFFFE //command value to be sent to play/pause
#define maxsong 4 //no.of songs on sd card
#define stop_play 0xFFFF// command value to stops all playback
#define zerovol 0xFFF7 // max volume
bool time;
void delay(uint16_t time_needed);//generates delay using timer
void resetModule(); //toggles reset pin to reset the module
void sendCommand(unsigned int command);// sends command patterns through data pin of module
void TA0_0_IRQHandler(void) ; //timer isr routine
#include "mp3board.h"
//ADAPTED FROM SPARFUNS ADRUINO FIRMWARE EXAMPLE:https://github.com/sparkfun/Audio-
Sound_Breakout-WTV020SD/blob/V_1.0/firmware/Audio-Sound_Breakout/AudioModule.ino
//Code written for msp432 by Sharan Arumugam and Atharv Desai
//DELAY FUNCTION USING TIMER A0
void delay(uint16_t time_needed)
{

  time=false;
  TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;
```

```
    // TACCR0 interrupt enabled
    TIMER_A0->CCR[0] = 2*time_needed;
    TIMER_A0->CTL = TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__UP; // SMCLK, UP mode

    TIMER_A0->CCTL[0] = TIMER_A_CCTLN_CCIE;
    __enable_irq();
    NVIC->ISER[0] |= 1 << ((TA0_0_IRQn) & 31);
    while(time==false)
       ;
}

void resetModule()
{

    clearrstpin;
          //500 ms delay function
    delay(15700);
    delay(15700);
    delay(15700);
    delay(15700);
    delay(15700);
    delay(15700);
    delay(15700);
    delay(15700);
    delay(15700);
    delay(10000);
    setrstpin;
    delay(15700);
    delay(15700);
```

```
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(10000);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(10000);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(15700);
delay(10000);
delay(15700);
```

```
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(10000);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(15700);
   delay(6800);
}
void sendCommand(unsigned int command)
{

   clearclkpin;
   delay(2950);//corressponds to 1900 us delay
   uint8_t i;
   for (i = 0; i < 16; i++)
   {
      delay(110);// corresponds to 100 Microseconds delay
```

```
      clearclkpin;
      clearcommandpin;
      if ((command & 0x8000) != 0)
      {
         setcommandpin;
      }
      delay(110);// corresponds to 100 Microseconds delay
      setclkpin;
      command = command<<1;
   }
}


void TA0_0_IRQHandler(void) {
   // Clear the compare interrupt flag
   TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;
   time=true;                    //setting flag
   TIMER_A0->CTL =TIMER_A_CTL_MC__STOP; //stopping timer
   TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIE; //disabling interrupt
}
//contains uart initalisation functions and polling driven transmit and receive functions
//Authors Sharan Arumugam and Atharv Desai
#include <stdbool.h>
#include <stdint.h>
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>


void uart2_init(void);
void uartinit_0(void);
```

```
int getcharBT(void);//receive uart signal
#include "uart.h"
//contains uart initalisation functions and polling driven transmit and receive functions
//Authors Sharan Arumugam and Atharv Desai
//BAUD RATE FOR BOTH SET AT 9600
//values taken from TIS online calculator http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430BaudRateConverter/index.html
//not used sets up uart communication to pc port
const eUSCI_UART_ConfigV1 uartConfig0 =
{
 EUSCI_A_UART_CLOCKSOURCE_SMCLK,        // SMCLK Clock Source
 19,                        // BRDIV = 19
 8,                         // UCxBRF = 8
 0,                         // UCxBRS = 0
 EUSCI_A_UART_NO_PARITY,            // No Parity
 EUSCI_A_UART_LSB_FIRST,            // LSB First
 EUSCI_A_UART_ONE_STOP_BIT,         // One stop bit
 EUSCI_A_UART_MODE,                 // UART mode
 EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION,  // Oversampling
 EUSCI_A_UART_8_BIT_LEN             // 8 bit data length
};

const eUSCI_UART_ConfigV1 uartConfig2 =
{
 EUSCI_A_UART_CLOCKSOURCE_SMCLK,        // SMCLK Clock Source
 19,                        // BRDIV = 78
 8,                         // UCxBRF = 2
 0,                         // UCxBRS = 0
 EUSCI_A_UART_NO_PARITY,            // No Parity
```

```
 EUSCI_A_UART_LSB_FIRST,              // LSB First
 EUSCI_A_UART_ONE_STOP_BIT,           // One stop bit
 EUSCI_A_UART_MODE,                   // UART mode
 EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION,  // Oversampling
 EUSCI_A_UART_8_BIT_LEN               // 8 bit data length
};
void uartinit_0(void)
{

   /* Selecting P1.2 and P1.3 in UART mode */
   MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
                                  GPIO_PIN2 | GPIO_PIN3, GPIO_PRIMARY_MODULE_FUNCTION);



   //![Simple UART Example]
   /* Configuring UART Module */
   MAP_UART_initModule(EUSCI_A0_BASE, &uartConfig0);

   /* Enable UART module */
   MAP_UART_enableModule(EUSCI_A0_BASE);
   MAP_UART_enableInterrupt(EUSCI_A0_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT);
   MAP_Interrupt_enableInterrupt(INT_EUSCIA0);



}

void uart2_init(void)
{
```

```
    MAP_GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P3,
                                    GPIO_PIN2 | GPIO_PIN3, GPIO_PRIMARY_MODULE_FUNCTION);




    //![Simple UART Example]
    /* Configuring UART Module */
    MAP_UART_initModule(EUSCI_A2_BASE, &uartConfig2);

    /* Enable UART module */
    MAP_UART_enableModule(EUSCI_A2_BASE);



}

int getcharBT(void)
{
    while(!(EUSCI_A2->IFG & EUSCI_A_IFG_RXIFG));
    return EUSCI_A2->RXBUF;
}

// Main routine for Audio player module with Bluetooth interface and control over playback
//Sharan arumugam and Atahrv Desai - Authors

/* DriverLib Includes */
//
#include "mp3board.h"
/* Standard Includes */
```

```
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include "uart.h"
#include "hd44780.h"



int getchar(void);
int putchar(int _x);




int main(void)
{
   /* Halting WDT  */
   MAP_WDT_A_holdTimer();

   P2->DIR |= BIT2;
   P5->DIR |= BIT1;//commandpin
   P6->DIR |= BIT1;//clkpin
   P5->DIR |= BIT6;//rstpin
   P2->DIR &=~BIT6;//INPUT BUSY PIN
   uartinit_0();//for pc uart
   uart2_init();//for bt uart
```

```
resetModule();
lcdinit();

uint8_t songnumber;
uint8_t volume;



P2->DIR |= BIT2;



char menuselect; //contains received uart value and function flow is governed by received value
while(1)
{



   menuselect=getcharBT();
   //printf("Received %c\n",k);
   if(menuselect=='1') //plays first song
   {
      lcdputstr("Playing song",0);
      printf("Playing song\n\r");
      resetModule();
      sendCommand(0);
      sendCommand(0xFFFE);

   }
   else if(menuselect=='2') //stop playback
```

```
{
   sendCommand(stop_play);
   printf("Stopping playback\n");
   lcdclear();
   lcdputstr("Stopping playback",0);
}
else if(menuselect=='3')//playnextsong
{
   sendCommand(stop_play);
   if(songnumber<=maxsong)
      songnumber++;
   resetModule();
   sendCommand(songnumber);
   sendCommand(play_pause);
   printf("Playing next song\n\r");
   lcdclear();
   lcdputstr("Playing next song",0);



}
else if(menuselect=='4')//
{
   sendCommand(play_pause);
   printf("Play/Pause \n\r");
   lcdclear();
   lcdputstr("Play/Pause",0);
}
else if(menuselect=='5') //HARD RESET
{
```

```
    resetModule();
    songnumber=0;
    lcdclear();
    lcdputstr("Hard Reset",0);
  }
  else if(menuselect=='6') //volume change up
  {
    volume++;
    if(zerovol|volume > 0xFFF7)
    {
      volume--;
      printf("Max Volume reached\n\r");
      lcdclear();
      lcdputstr("Volume increased ",0);
    }
    else
      sendCommand(zerovol|volume);
  }
  else if(menuselect=='7')//volume change down
  {
    volume--;
    if(zerovol|volume <0xFFF0)

    {
      volume++;
      printf("Min Volume reached\n\r");
      lcdputstr("Play/Pause",0);
    }
    else
```

```
            sendCommand(zerovol|volume);
        }



        if(P2->IN &BIT6)
           printf("Songs playing\n\r");


    }

}

int getchar(void)
{
    while(!(EUSCI_A0->IFG & EUSCI_A_IFG_RXIFG));
    return EUSCI_A0->RXBUF;
}

int putchar(int _x)
{
    while(!(EUSCI_A0->IFG & EUSCI_A_IFG_TXIFG));
    EUSCI_A0->TXBUF = _x;
    return 0;
}
```