CpE5160 Experiment #3
The I2C interface and STA013 MP3 decoder

Introduction

The purpose of this experiment is to create the functions needed for the Inter-Integrated Circuit (I2C) bus and to use these functions to initialize the STA013 MP3 decoder. The STA013 MP3 decoder schematic is given in this handout and this circuit will need to be added to the 8051 circuit. Two general purpose I/O pins need to be selected to be the SCL and SDA signals for the I2C bus. Another general purpose I/O pin will be needed to control a reset pin on the STA013. A fourth general purpose I/O pin will be needed as an input for the data request output from the STA013. A fifth general purpose I/O pin is used as a select pin for the SPI compatible communication used to send data to the STA013. The I2C functions for reading and writing to a device will need to be developed. These functions will manually create the signals required for the clock (SCL) and data (SDA) for proper transmission. These functions will then be used to transmit the values to configure the STA013.

The STA013 MP3 decoder circuit

The STA013 MP3 decoder is a digital signal processor that is preprogrammed with an MP3 decoding algorithm. It operates on 3.3V so it can be connected to the same power supply as the 8051 and SD card. There are four connections to 3.3V and each one should have a 0.1uF bypass capacitor. There is a separate supply connection for the phase lock loop (PLL). These supply connections should be connected to 3.3V and ground through small value resistors (2 to 5 ohms) along with a bypass capacitor. The STA013 has its own oscillator. The clock for the processor comes from a phase lock loop that multiplies the output of the oscillator. The PLL filter circuit is connected to the filter pin which consists of a 470pF capacitor in parallel with a series combination of a 1K resistor and a 0.0047uF capacitor. Be careful to use as short of connections as possible for the power supply, oscillator and PLL filter connections. There is a RESET pin that must be driven low for at least 100 nsec. This pin can be connected to a general purpose I/O pin that can be switched low to reset the device. There are two pins that are used for manufacturing purposes and need to be disabled. The nTESTEN pin needs to be connected to 3.3V and the SCANEN pin needs to be connected to ground. There is also a pin unused in this circuit, SCA_INT, that must be tied to 3.3V. The rest of the pins are used for communication.

Communication with the STA013

The Inter-Integrated Circuit (I2C) bus is used to write data to the registers to configure the STA013. The two pins for the I2C bus are the SCL pin for the clock and the SDA pin for the data. Both of these pins should be connected to 3.3V through 4.7K resistors. The bus is idle when both of the pins are high. Communication begins with the start condition when the SDA pin is switched low while the SCL pin is high. When sending data bits, the SDA pin should not be changed unless the clock is low. When the SCL pin is switched high, it should be checked to ensure that it is high. This is because slower devices can hold the clock low until they are ready. The SDA pin is sampled for its data value when SCL is high. The SDA pin needs to be checked by the master to make sure

that it is the same value that it has asserted on the bus. If it is not, then another master is communicating on the bus and the first master must stop transmitting. The first byte transmitted is always sent by the master and it is the device address. The device address is a 7-bit value. The least significant bit (lsb) of the first transmitted byte is a read or write bit. A '1' indicates a read and a '0' indicates a write. The slave device should respond with an acknowledge (ACK) which is a '0.' The master must set the SDA pin high and provide a clock cycle for the ACK from the slave. If the transmission is a write, then the next byte is sent by the master and is typically the sub-address or internal address of the register to be written. The slave device should respond with an ACK after each byte that is sent. The master can then send a data byte to be written to the register. The transmission ends with a stop condition. This is a transition of the SDA pin from low to high while the SCL pin is high.

If the transmission is a read, then the slave will respond with a data byte after the device address byte (first byte) is sent by the master and the ACK is sent by the slave. The SCL pin is cycled low and then high by the master for each bit to be received. The master must set the SDA pin high to allow the slave to control the data value. The master should check the SDA pin after the SCL pin is verified to be high to read the incoming bit. The master responds with an acknowledge (ACK or '0') or a not acknowledge (NACK or '1') after each byte. The master should respond with an ACK if more data bytes are expected or a NACK if this is the last byte. At the end of the transmission a stop condition must be sent. The STA013 data sheet has a brief description and a diagram of the I2C communication types.

The data values of the MP3 file are sent to the STA013 using an interface similar to the SPI interface used to communicate with the SD card. One difference is that the names are different than the typical SPI interface (SCKR instead of SCK and SDI instead of MOSI). Another difference is that there is no MISO equivalent pin, so the communication is one way. If the SCKR polarity is set correctly, then the SPI peripheral can be used to communicate with both the SD card and the STA013. The BIT_EN pin operates like the nCS pin on the SD card, except that it is of the opposite polarity. In other words, the STA013 will ignore any SPI communication unless BIT_EN is high and the SD card will ignore any SPI communication unless nCS is low. The programmer may wish to create their own SPI communication for the STA013 to help with keeping the data flow continuous between the SD card and STA013. With this setup the 8051 can pause the data transfer from the SD card as needed to send data to the STA013. This interface can be created by sending out each byte most significant bit (msb) first. The data bit must be changed on one edge of the clock and remain stable on the other edge. This edge is determined by the setting of the SCKR polarity bit.

The final set of communication pins are used to output the decoded data to a digital to analog convertor (DAC). The CS4334 used in this project requires an I2S interface. This is configured in the STA013. When the system is operating correctly while outputting decoded data, the LRCKT output will have a frequency equal to the sample rate of the MP3 file (typically 44.1 KHz for 192Kbps songs, 16KHz for 30Kbps songs and 8KHz for 10Kbps songs). The CS4334 is fairly flexible on the oversampling rate and the number

of data bits that can be sent. See the datasheet for all of the options. The output of the CS4334 is a line level signal and it can be connected to amplified speakers. Use the 3.5mm connector for this connection. The black wire is the common and the red and green wires are the right and left connections.

STA013 Operating Modes

The STA013 operates in two different modes, multimedia mode and broadcast mode. In multimedia mode, a buffer in the STA013 is used to store values until the DSP needs them. The DATA_REQ pin is used to signal when the buffer needs more values. This pin can be configured to be active high or active low. In broadcast mode, the data must be sent at a bit-rate that is equivalent to the bit-rate of the MP3 file. In our system, the multimedia mode is a better fit. The 8051 can read values from the SD card into XRAM and then write those values into the STA013 when the DAT_REQ pin is active. The only requirement is that there is enough time between data requests to load more data from the SD card. The bit-rate of the MP3 file determines how fast the data must be transferred to the STA013. The files in the root directory of the SD card are at a slower bit-rate of around 30Kbps or less and the finished system should be able to play these songs. A faster system may be needed to play the faster songs on the SD card. The SPI clock rate determines how fast a data block can be downloaded from the SD card. Using an 18.432MHz crystal and the x2 mode gives us the fastest possible transfer rate for our system.

The Configuration (Patch) File

A configuration or patch file was recommended by several STA013 users. The exact purpose of this file is unknown, but I found by experimentation that it is required. An assembly file containing the configuration file can be found on Canvas. Each line of the file has two bytes. The first is a register address and the second is the data value to write to that register. This file is split into two sections. A write to the last register of the first section causes a software reset. A short pause of at least 100nsec should occur after this write. Each section is terminated with 0xFF in both the register address and data. This allows the program to write values until the 0xFF is read from the file. This is much easier than counting how many values have been written since there are over 2,000 values in this file. Both sections of the file have a label to mark the start of the section and these labels are declared as public to make them available to other files. They may be declared as “extern uint8_t code” values in the C source code. The address of these values can be used as pointers to the first line of each section.

STA013 Configuration

The configuration flow given on page 31 of the STA013 data sheet shows what registers need to be written to configure the device. The first values are for the pulse code modulator which creates the signals for the DAC. I chose to set up the device for an I2S output (msb first, right padded, I2S compliant, data sent on falling edge). The number of bits and oversampling ratio is up to the student. I assumed that 16-bit resolution and 512 oversampling ratio was sufficient and chose those settings, but other settings should work as well. After these selections are made, then the charts on pages 32 and 33 can be used to select the next set of values. Since the oversampling ratio I used was 512 and the

crystal frequency was 14.7456MHz, I used the values from table 11. After the values from the table are written, then set the SCKR polarity (0x0D) to match our existing SPI polarity (data stable on the rising edge, POL=0). Next enable the DATA_REQ pin (0x18) and set the active polarity (0x0C). From my experience, the audio output will clip if it is not attenuated a little bit in the STA013. I wrote a value of about 7 to 10 to the DLA (0x46) and DRA (0x48) registers to get this attenuation. The final step is to write a 1 to the Run (0x72) register. If all is operating correctly, then the DATA_REQ signal should be asserted indicating the MP3 decoder is waiting on data.

Procedure

The following equipment will be required for this part of the experiment:

- Microcontroller circuit with SD card from Exp. #2
- STA013 MP3 decoder with associated components
- CS4334 DAC with associated components
- 9V battery or external power supply
- Serial Cable (check out from 210EECH)
- STA013 configuration data file (STA013_Config.asm)
- PC with
    - Keil uVision2 or later
    - F.L.I.P.
    - Terminal Program (Putty)

You should create two source code and header file pairs for this experiment. The first source code and header file should be for the I2C_Write and I2C_Read functions (including any delay or private functions created for these functions). Since these are generic functions that could be used with any I2C device, write them as if you plan to reuse them in another project. The second source code and header file will be for the functions used to initialize the STA013. It will use the I2C functions to write the patch file and your configuration data to the STA013.

1) Wire up the MP3 decoder circuit that is included with this handout. The CS4334 DAC does not need to be connected at this time, but it can be if so desired. Note that the CS4334 requires a +5V supply.

The I2C_Read and I2C_Write functions should be in their own source code file with their prototypes in a header file so they can be called by other source code files. This allows them to be reused in other projects that need I2C communication.

2) (Functionality: 15pts) Create a read function for the I2C interface. It will have five input parameters: the address of the device from which data will be read, the internal address (if needed), the internal address size (0, if there is no internal address), the number of bytes to read and a pointer to an array where the data bytes can be stored. The function will return an error value to indicate if the read was successful or not. The data that is read will be placed in the array.

The recommended prototype for this function is:
*uint8_t I2C_Read(uint8_t device_addr, uint32_t int_addr, uint8_t int_addr_sz, uint8_t num_bytes, uint8_t * array_name);*

The STA013 MP3 decoder datasheet has descriptions of a read sequence (figure 11 on page 10). Note that a read operation does not send a register address to indicate from where the data will be read. Instead a write operation must be used first to set the register address. Since the I2C_Write function has not been written yet, the int_address and int_addr_sz parameters will not be used yet. The I2C_Read function then performs the following steps: Create a start condition, send the device address with a 1 in the lsb, receive an ACK from the slave, receive a data byte from the slave. If only one byte is to be received, then send an NACK to the slave and create a stop condition. If more than one byte is to be received, then an ACK is sent after all but the last byte that is received.

3) (Debugging) Create a function that will eventually be used to initialize the STA013. It can be in its own source code file (recommended) or be a part of the main function. It will be called from main during the system initialization. At this point, the function will only call the I2C_Read function for debugging. The internal register pointer of the STA013 after power up should be pointing at the last internal register. Therefore reading three bytes should read the last internal register and then the first two internal registers. The second internal register (third byte that is read) is an ID register with a fixed value of 0xAC. Call the I2C_Read function with a device address of 0x43 (found in the STA013 datasheet) to read three bytes. The internal address and internal address size are not used and can be 0. Print the third byte that was read and verify that it is 0xAC. I2C functions can be called in a loop so that if the slave device is busy and does not respond the first time, then the I2C_Read function can be attempted again.

   The sequence might look something like this:

```
i=timeout_val;  // This value is the number of attempts
do
{
    error=I2C_Read(0x43,0,0,3,array_name);
    i--;
 }while((error!=no_errors)&&(i!=0));
  printf("Received Value = %2.2bX\n\r", array_name[2]);
```

   Please leave this in the code for grading. It can be commented out after you have verified that it works.

4) (Functionality: 10pts) Create a write function for the I2C interface. It will have five input parameters: the address of the device to which data will be written, an internal address value (if needed), the internal address size (0, if there is no internal address), the number of bytes to write and a pointer to an array of data bytes. The function will return an error value to indicate if the write was

successful or not. The recommended prototype for this function is:
*uint8_t I2C_Write(uint8_t device_addr, uint32_t int_addr, uint8_t int_addr_sz, uint8_t num_bytes, uint8_t * array_name);*

The STA013 MP3 decoder datasheet has descriptions of a write sequence (figure 10 on page 10). The sequence is: create a start condition, send the device address with a 0 in the lsb, receive an ACK from the slave, send the register address, receive an ACK from the slave, send the data which goes to that register, receive an ACK from the slave, create a stop condition.

5) (Functionality: 5pts) The I2C_Write function can be tested by calling it before or from within the I2C_Read function to set the internal address to read from. Use the I2C_Write function to set the internal address to 0x01. Then use the I2C_Read function to read the value from this address which again should be 0xAC. **Keep this read of the ID register for the demonstration of this experiment**.

The sequence might look something like this:

```
i=timeout_val; // Set the number of attempts
do
{
    // I2C_Write called from within I2C_Read
     error=I2C_Read(0x43,0x01,1,1,array_name);
    i--;
 }while((error!=0)&&(i!=0));
  printf("Received Value = %2.2bX\n\r", array_name[0]);
```

During testing, the student may wish to call the I2C_Write function separately so that one function can be debugged at a time. If the student wants to call the I2C_Write function and then the I2C_Read function for debugging, it may look something like this:

```
i=timeout_val;
do
{
    error=I2C_Write(0x43,0x01,1,0,array_name);
    i--;
 }while((error!=0)&&(i!=0));
 i=timeout_val;
do
{
    error=I2C_Read(0x43,0,0,1,array_name);
    i--;
 }while((error!=0)&&(i!=0));
  printf("Received Value = %2.2bX\n\r", array_name[0]);
```

6) (Functionality: 5pts) The configuration or patch file from ST is available on Canvas as an assembly file. It is set up as two arrays of data bytes. Each array

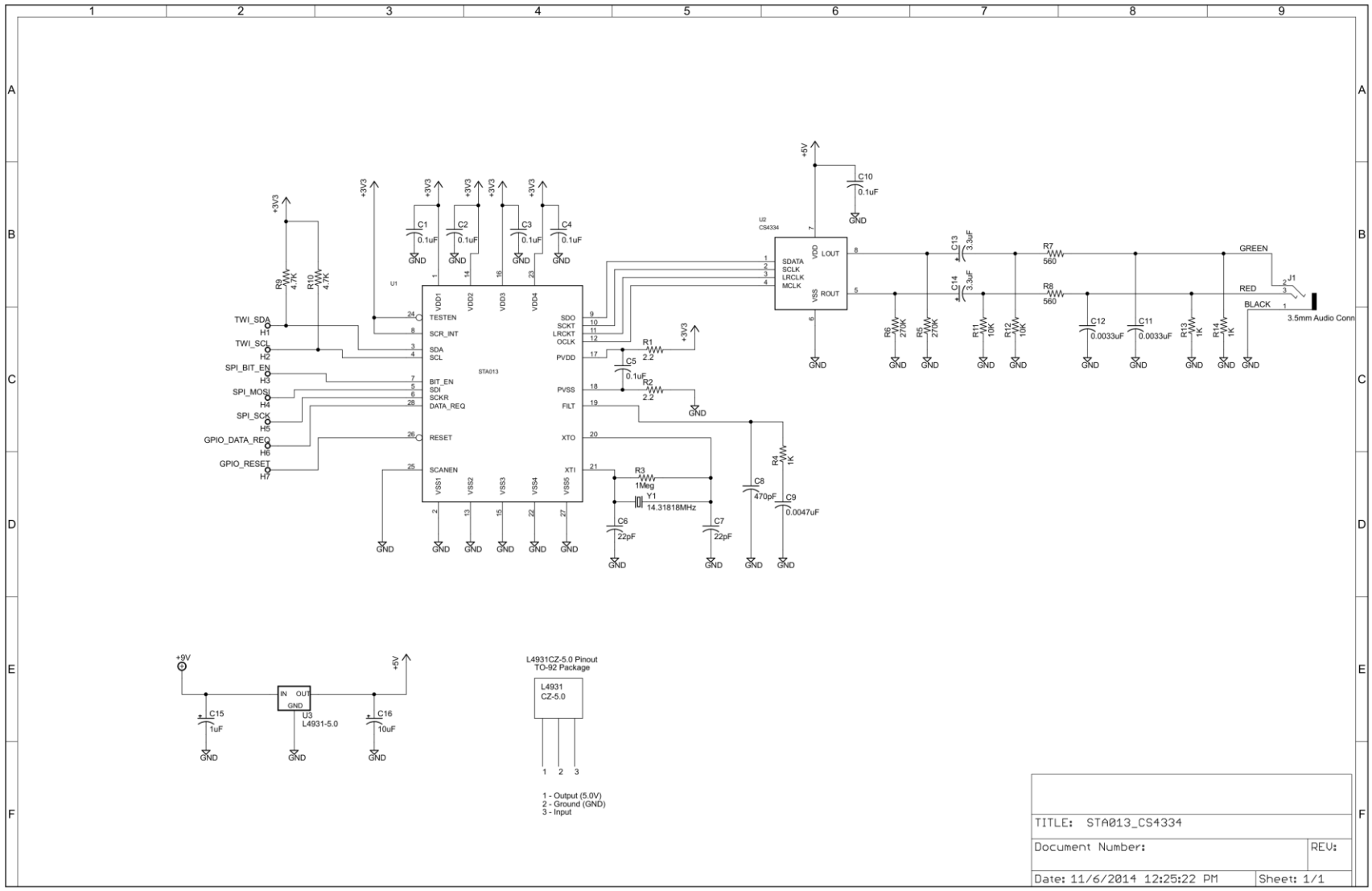
has a label to mark the starting point of the array in code memory. The array is organized as a register address and then the data for that register in successive addresses. Each array is terminated with four 0xFF values. The programmer can create a loop to continually write values to the STA013 until a 0xFF is detected. The first array ends with a write to the software reset register. A short pause should be placed in the software after this value is written. Note that writing all of these configuration values will take a few seconds, so be patient.

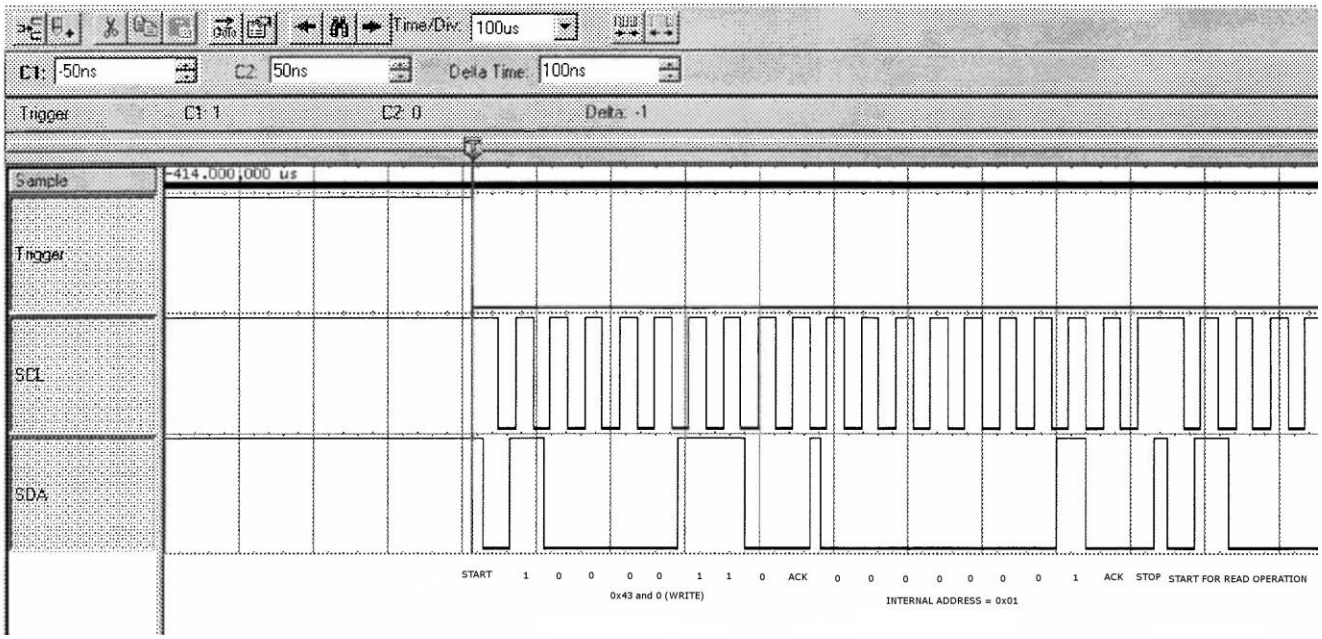
7) The programmer will also need to write the configuration values discussed in the datasheet which setup the I2S output and the PLL divider values which configure the device for the crystal frequency. These values can be added to the second part of the configuration file if the programmer wishes. The values written from the second part of the configuration file to configure the STA013 MP3 Decoder can be read from the registers they were written to and printed using the UART to verify that the values are being written correctly.

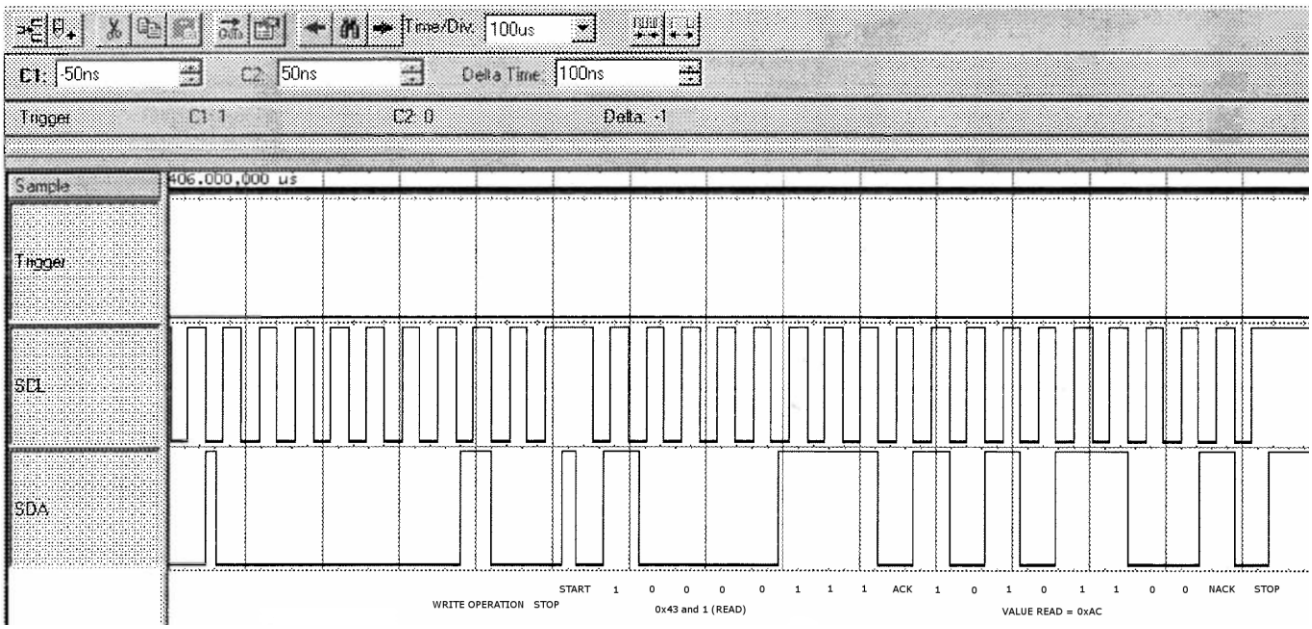
8) (Demonstration: 5pts) A demonstration of step 4, where you read the value 0xAC from the ID register will be required. I will use an oscilloscope or a logic analyzer to measure the SCL and SDA signals that are generated. A trigger signal in the code can be used to make the capture of the I2C signals easier. Use any spare I/O signal and clear it low right before the code given in step 4. Please remove any print statements that may be in between the trigger and the I2C_Write and I2C_Read functions. Print statements require a lot of execution time and will not allow both I2C communications to be captured.

Grading:

Functionality:	Code works as described in the assignment: Registers set as needed and actions occur as specified; Questions answered and calculations shown.	Points awarded as specified in the steps marked graded. (40 points)
Compile Errors:	If submitted code does not compile, then a minimum of 25% of the functionality points will be deducted. More may be deducted for multiple errors.	Minimum of 25% of functionality point deducted.
Organization:	Seperating device drivers (microcontroller, I2C), hardware drivers (STA013 config) and application code.	10 points
Readability:	Source code is well commented. Descriptive names are used for functions, constants and variables.	10 points
Correctness	No patches or work-arounds are used in the source code to get the correct functionality.	10 points


U1
STA013
VDD1
VDD2
VDD3
VDD4
TESTEN
SCR_INT
SDA
SCL
BIT_EN
SDI
SCKR
DATA_REQ
RESET
SCANEN
SDO
SCKT
LRCKT
OCLK
PVDD
PVSS
FILT
XTO
XTI
VSS1
VSS2
VSS3
VSS4
VSS5
U2
CS4334
VDD
VSS
SDATA
SCLK
LRCLK
MCLK
LOUT
ROUT
+3V3
+5V
GND
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10
C11
C12
C13
C14
0.1uF
22pF
470pF
0.0047uF
3.3uF
0.0033uF
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
2.2
1Meg
1K
4.7K
270K
560
10K
Y1
14.31818MHz
TWI_SDA
TWI_SCL
SPI_BIT_EN
SPI_MOSI
SPI_SCK
GPIO_DATA_REQ
GPIO_RESET
H1
H2
H3
H4
H5
H6
H7
GREEN
RED
BLACK
J1
3.5mm Audio Conn
+9V
+6V
U3
L4931-5.0
C15
C16
1uF
10uF
IN
OUT
GND
L4931CZ-5.0 Pinout
TO-92 Package
L4931
CZ-5.0
1 - Output (5.0V)
2 - Ground (GND)
3 - Input
TITLE: STA013_CS4334
Document Number:
REV:
Date: 11/6/2014 12:25:22 PM
Sheet: 1/1


Time/Div: 100us
C1: -50ns
C2: 50ns
Delta Time: 100ns
Trigger
C1: 1
C2: 0
Delta: -1
Sample
-414.000,000 us
Trigger
SCL
SDA
START 1 0 0 0 0 1 1 0 ACK 0 0 0 0 0 0 0 1 ACK STOP START FOR READ OPERATION
0x43 and 0 (WRITE)
INTERNAL ADDRESS = 0x01


Time/Div: 100us
C1: -50ns
C2: 50ns
Delta Time: 100ns
Trigger
C1: 1
C2: 0
Delta: -1
Sample
406.000,000 us
Trigger
SCL
SDA
START 1 0 0 0 0 1 1 1 ACK 1 0 1 0 1 1 0 0 NACK STOP
WRITE OPERATION STOP
0x43 and 1 (READ)
VALUE READ = 0xAC