CpE5160 Experiment #4
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 or Two-Wire Interface (TWI) and to use these functions to initialize the STA013 MP3 decoder. The STA013 MP3 decoder schematic is given in this handout for your reference. The TWI1 peripheral with SDA on PE5 and SCL on PE6 will be used for connecting to the STA013. Another general purpose I/O pin (PB1) is connected to the reset pin on the STA013. A second general purpose I/O (PC6) is connected to the DATA_REQ output from the STA013. A third general purpose I/O pin (PD6) is used as a select pin (BIT_EN) 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 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 is connected to the same power supply as the SD card. There are four connections to 3.3V and each has a 0.1uF bypass capacitor. There is a separate supply connection for the phase lock loop (PLL). These supply connections are 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. There is a RESET pin that must be driven low for at least 100 nsec. This pin is connected to a general purpose I/O pin (PB1) that can be switched low to reset the device. There are two pins that are used for manufacturing purposes and are disabled. The nTESTEN pin is connected to 3.3V and the SCANEN pin is connected to ground. There is also a pin unused in this circuit, SCA_INT, that is 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 pins are connected to 3.3V through 4.7K resistors as pull-up 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 is checked to ensure that it is high by the TWI module. 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 is 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 sets the SDA pin high and provides 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 that is written to the internal register. Multiple bytes can be sent and typically, the internal address will auto-increment for each byte written. 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 sets the SDA pin high to allow the slave to control the data value. 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 is 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 like 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 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 somewhat 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.

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 bitrate that is equivalent to the bitrate of the MP3 file. In our system, the

multimedia mode is a better fit. The ATMEGA324PB can read values from the SD card into SRAM 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 bitrate 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.

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. A C source code 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 in both register address and the data. This is much easier than counting how many values have been written since there are around 4,000 values in this file. Both sections of the file have a label to mark the start of the section and they are declared in a manner that places them in Flash memory. They may be declared as “extern const uint8_t CONFIG[3998]” and “extern const uint8_t CONFIG2[50]” in other C source code files to import the labels. The address of the first value of the array (&array_name[0]) can be used as a pointer to the first line of each section.

STA013 Configuration

The configuration flowchart 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 can be chosen by the programer. 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 12. 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) (I selected an active low polarity). 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 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:

- ATMEGA324PB development board
- STA013 MP3 decoder/CS4334 DAC board
- USB Cable
- STA013 configuration data file (Config_Arrays.c)
- IO1 Xplained expansion board (optional)
- PC with
    - Microchip Studio
    - 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 TWI_master_init, TWI_master_transmit and TWI_master_receive 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.

The TWI_master_init, TWI_master_transmit and TWI_master_receive 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.

1) (Functionality: 5pts) Create a function to initialize the TWI module by setting the TWI clock frequency. The function is passed a pointer to the module to use and a clock frequency for the TWI module. This should be used to determine the prescaler value and baud rate generator value. These values are then written to the appropriate registers. As an option, this clock initialization can be included into the TWI_master_transmit and TWI_master_receive functions.

   The recommeneded prototype for this function is:
   *uint8_t TWI_master_init(volatile TWI_t *TWI_addr, uint32_t I2C_freq);*

2) (Functionality: 10pts) Create a read function for the I2C interface. It will have four or six input parameters: The address of the TWI module to use (TWI1), the address of the slave device (0x43), the internal address (optional, implemented later, bonus), the internal address size (optional, implemented later, bonus), 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 TWI_master_receive(uint8_t volatile *TWI_addr, uint8_t device_addr, uint32_t int_addr, uint8_t int_addr_sz, uint16_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 TWI_master_transmit function has not been written yet, the int_address and int_addr_sz parameters will not be used yet. The TWI_master_transmit function has a bonus part to have these values as separate parameters and sending them before the data is sent.

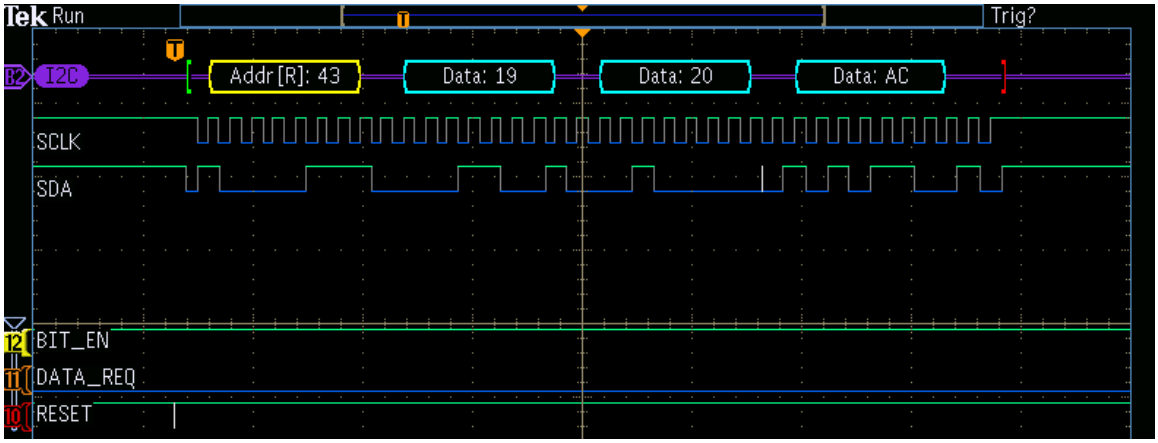
The TWI_master_receive then performs the following steps: Create a start condition, send the device address with a 1 in the lsb, receive an ACK from the slave, receive a data byte from the slave. If only one byte is to be received, then send a NACK to the slave and create a stop condition. If more than 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 should be in its own source code file. It will be called from main during the system initialization. At this point, the function will only call the TWI_master_receive function for debugging. Before the first access to the STA013, apply a reset signal by pulsing PB1 low for at least 100nsec. The internal register pointer of the STA013 after applying a low pulse to the reset pin 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 TWI_master_receive function with a device address of 0x43 (found in the STA013 datasheet) and using TWI1 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 (172 decimal). 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 TWI_master_receive function can be attempted again.

The sequence might look something like this:

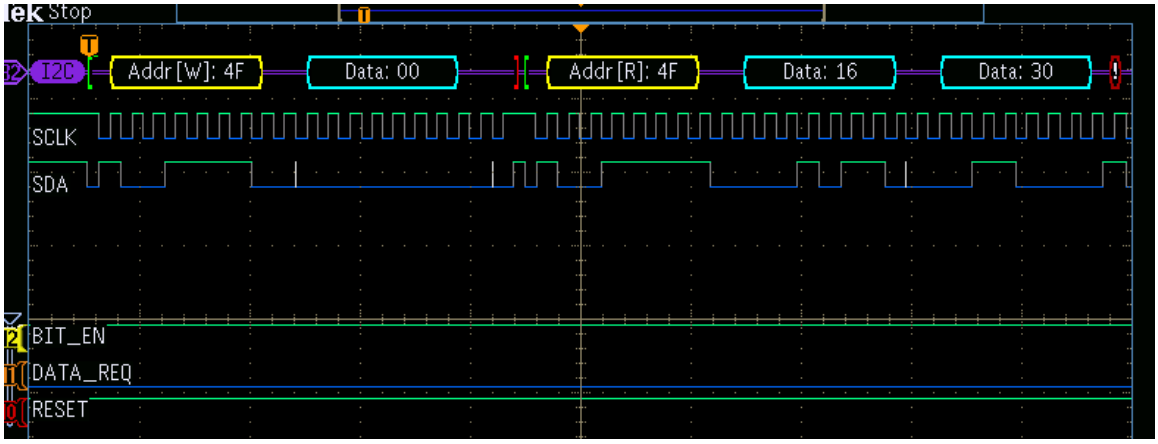
```
i=timeout_val;  // This value is the number of attempts
do
{
    error=TWI_master_read(TWI1,0x43,0,0,3,array_name);
    i--;
 }while((error!=no_errors)&&(i!=0));
  sprintf(prnt_bffr, "Received Value = %2.2bX\n\r", array_name[2]);
  UART_transmit_string(UART1,0,prnt_bffr);
```

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



This image shows an I2C read of three bytes following a reset of the STA013. In this case the reset was used as the trigger for the oscilloscope (edge trigger: D10). You can also trigger on the I2C start condition (Bus trigger: trigger on: START). The goal is to read 0xAC on the third byte. The values on the first and second bytes may be different. The green [indicates a start and the red] indicates the stop. The device address (0x43) is in yellow and the R indicates an I2C read.

Optional method for debugging is to read the temperature value, which is the default internal register, from the AT30TSE758 on the IO1 Xplained expansion board. The slave device address is: 0x4F (0b1001111). The value is a two-byte value (integer and fraction) temperature reading in Celsius. The expected value for the integer portion is around 0x16 (22C or 71F).



This image is an I2C read of the temperature from the AT30TSE758 on the IO1 expansion board. It shows a write to set the address to 0x00 and then a read, but the write is not needed if no other accesses to the AT30TSE758 have occurred. Note the stop and start that occurs between the I2C write and I2C read. Also note the ! in red at the upper right of the picture. This is indicating the NACK at the end of the read (SDA=1 after the data byte) and that no stop was detected (it occurred off screen).

4) (Functionality: 10pts, bonus 5pts) Create a write function for the I2C interface. It will have four or six input parameters: the address of the TWI module to use, the address of the slave device, an internal address value (optional, bonus), the internal address size (optional, bonus), 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 TWI_master_transmit(uint8_t volatile *TWI_addr, uint8_t device_addr, uint32_t int_addr, uint8_t int_addr_sz, uint16_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 internal register address (1 to 4 bytes), receive an ACK from the slave, send the data which goes to that register, receive an ACK from the slave, multiple bytes could be sent and the internal address will usually auto-increment, and then create a stop condition. If you choose not to do the bonus part, the internal address will be passed as the first byte(s) of array_name instead of being passed as a separate parameter. The internal address is then sent as any other data byte(s).

   The optional internal address and internal address size is a bonus part (worth 5pts) to have a separate code section that sends the internal register address before the data is sent. The internal address can be up to 32-bits and the int_addr_sz indicates the number of bytes of the internal address (0, 1, 2, 3, or 4 bytes). The int_addr_sz parameter is used to split up the internal address into bytes and send them most significant byte (MSB) first.

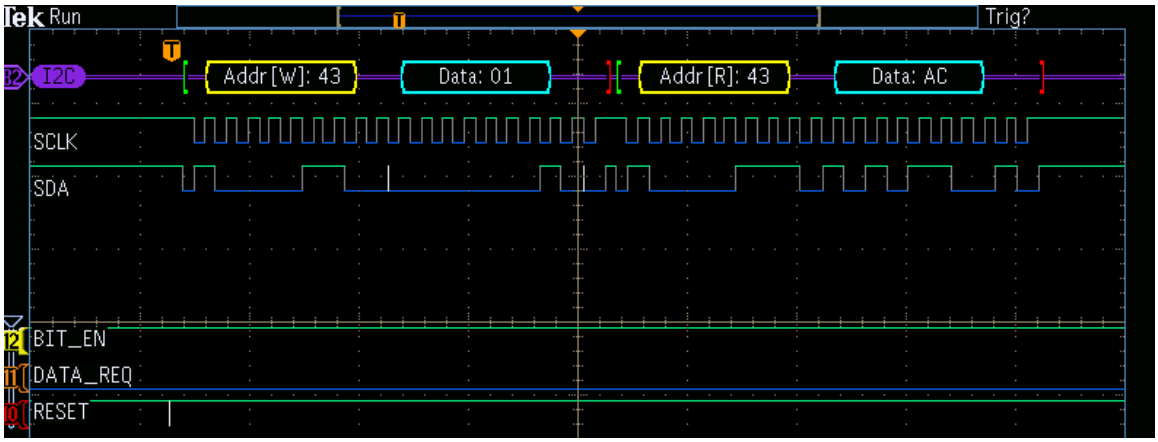
5) (Functionality: 5pts) The TWI_master_transmit function can be tested by calling it before or from within the TWI_master_receive function to set the internal address to read from. Use the TWI_master_transmit function to set the internal address to 0x01. Then use the TWI_master_receive function to read the value from this address which again should be 0xAC. Before the first access to the STA013, apply a reset signal by pulsing PB1 low for at least 100nsec. Keep this read of the ID register for grading.

   The sequence might look something like this:

```
i=timeout_val; // Set the number of attempts
do
{
    // TWI_master_transmit called from within TWI_master_receive
     error=TWI_master_read(TWI1,0x43,0x01,1,1,array_name);
    i--;
 }while((error!=0)&&(i!=0));
 sprintf(prnt_bffr, "Received Value = %2.2bX\n\r", array_name[0]);
 UART_transmit_string(UART1,0,prnt_bffr);
```

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

```
i=timeout_val;
do
{
    // Internal address set, no data bytes sent
    error=TWI_master_transmit(TWI1,0x43,0x01,1,0,array_name);
    i--;
 }while((error!=0)&&(i!=0));
 i=timeout_val;
do
{
    // Read from the internal address set by transmit
    error=TWI_master_receive(TWI1,0x43,0,0,1,array_name);
    i--;
 }while((error!=0)&&(i!=0));
 sprintf(prnt_bffr, "Received Value = %2.2bX\n\r", array_name[0]);
 UART_transmit_string(UART1,0,prnt_bffr);
```



This image shows the I2C write being used to set the internal address to 0x01 (only a one byte internal address) and no data being transmitted. This is followed by an I2C read of one byte. The data byte is followed by a NACK and a stop.

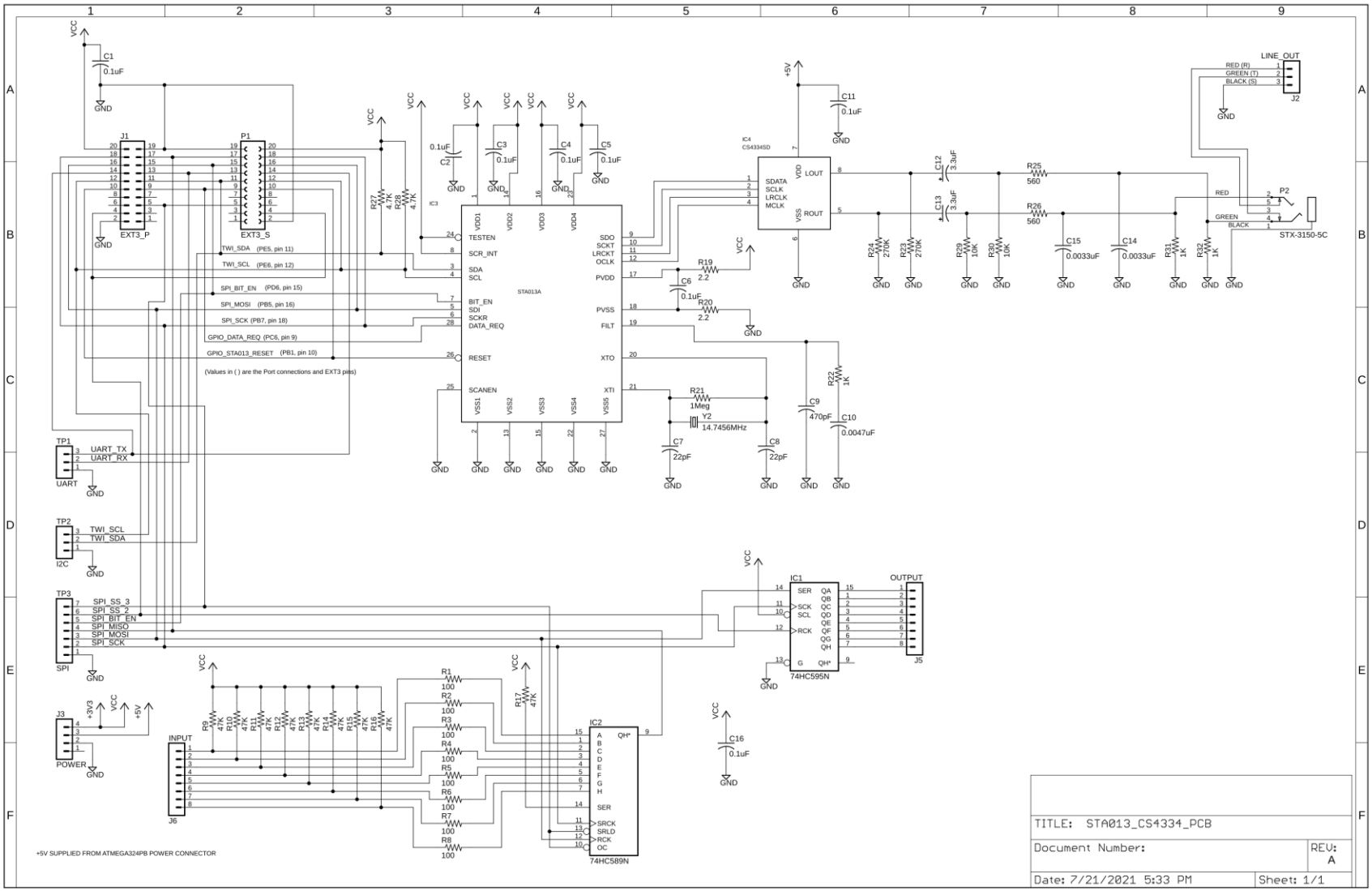
6) (Functionality: 5pts) The configuration or patch file from ST is available on Canvas as a C source code file. It is set up as two arrays of data bytes. Each array has a label to mark the starting point of the array in code memory. The array is organized as a register address and then the data for 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 in both the internal address and data. 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) (Functionality: 5pts) The programmer will also need to write the configuration values discussed in the datasheet which set up 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. I chose to create a third separate configuration file. The values written from this 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. If all three configuration files are written correctly, the DATA_REQ signal should be active.

Grading:

Functionality:	Code works as described in the assignment: Registers set as needed and actions occur as specified.	Points awarded as specified in the steps marked graded. (40 points)
Bonus:	Uses the optional parameters of internal address and internal address size in both TWI_master_transmit and TWI_master_receive	(5 bonus 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:	Separating 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 workarounds are used in the source code to get the correct functionality.	10 points


TITLE: STA013_CS4334_PCB
Document Number:
REV: A
Date: 7/21/2021 5:33 PM
Sheet: 1/1
(Values in ( ) are the Port connections and EXT3 pins)
+5V SUPPLIED FROM ATMEGA324PB POWER CONNECTOR
STA013A
CS43345D
74HC595N
74HC589N
STX-3150-5C