Design Project ELEC2117 17s1 – Matthew Notarangelo z5116928

Description of the Design Project

The aim of the project was to build an FM radio using the Si4703 FM radio module, a 16x01 LCD, and a keypad. This was all connected using a PIC16F886 microcontroller. The radio should have at least five tuned stations, a seek up button, a seek down button, and any other functions of my choosing. I decided to implement a mute button, and volume up/down buttons, an "LCD off" button, as well as other radio stations. The current channel frequency should be shown on the LCD and the radio module is to be programmed using the I2C protocol. This project is to be achieved using the assembly programming language.

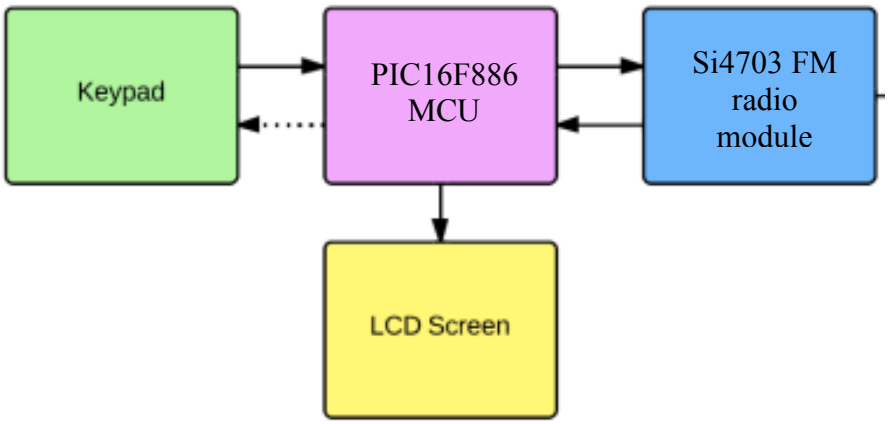
Design Criteria

The design criteria that I chose to base my project on were explicitly based on the project specification. This meant that the design had to work as per the given instructions and must work without fault. As such, the keypad should be interfaced so that the button pressed runs a function (ie. Seek up/down, go to channel, etc). It must work consistently, and without error. Whenever a button is not pressed it should keep waiting for a button to be pressed, and do nothing in the meantime. The LCD should be interfaced so as to show the current channel frequency. It must show the correct frequency, be presented on the LCD in a readable format, and must show the characters clearly. The LCD should also have its screen reset after each change in frequency so that the current frequency is the only thing shown.

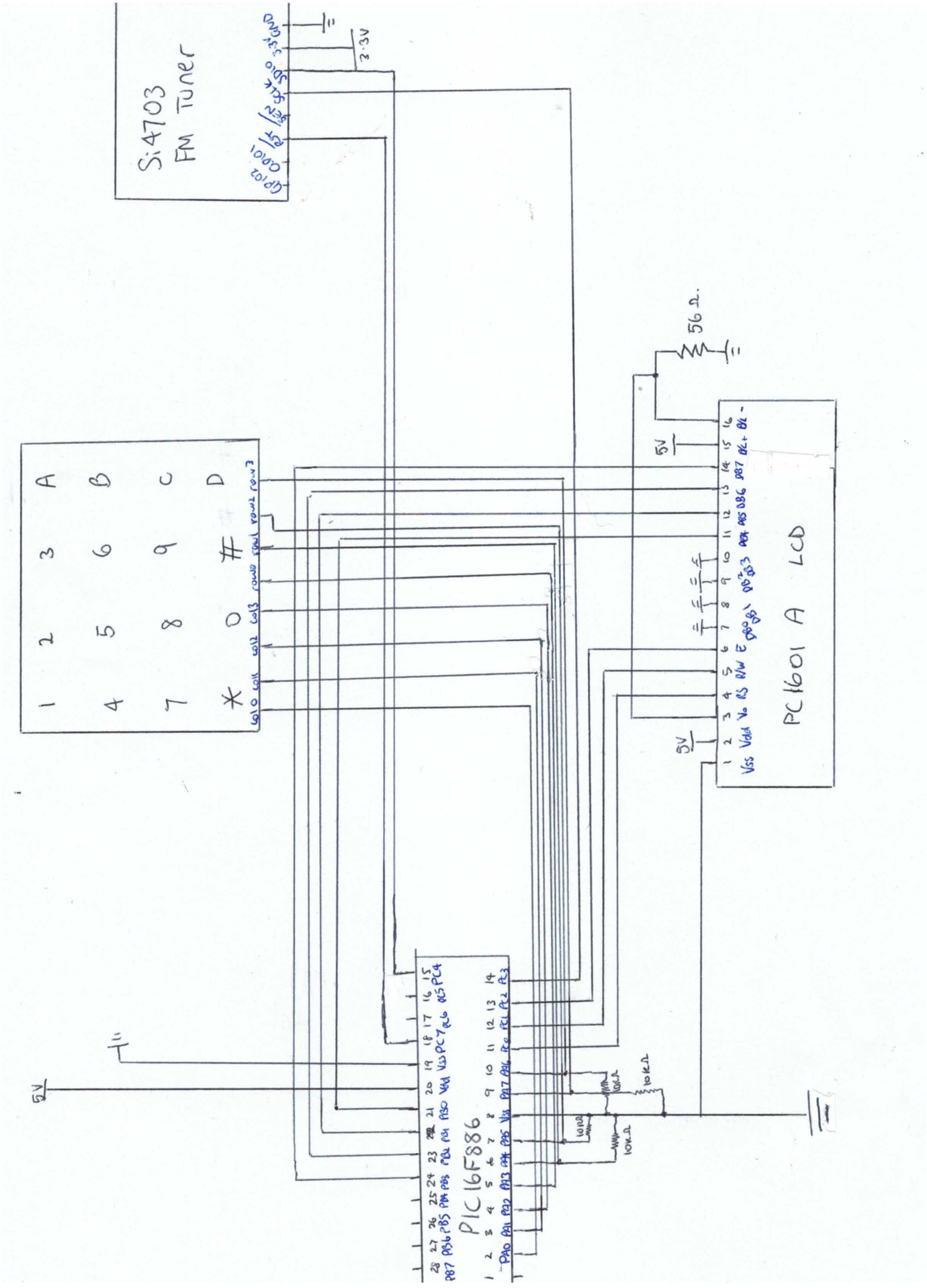
The FM module must receive instructions from the PIC microcontroller reliably using the I2C protocol. As such, it must not give any errors or fail. It must be able to change the station to preset frequencies, seek up and down, and have any other functions which could be useful. It must also be able to be written to and read from so as to show the frequency on the LCD. The main function is to receive an FM radio signal.

The PIC16F886 microcontroller must also work reliably and without error. It must be capable of interfacing all the other components and be able to initialise them all. Errors from any peripherals should be processed and handled accordingly. The code must also be efficient.

Block Diagram



Circuit Diagram



Details of Software Design

The software was designed with the design criteria in mind, in addition to reliability, length, readability, and complexity. I started by initializing the ports of the PIC microcontroller. This included setting the ports to be digital outputs, and some digital inputs. I also decided to initialise some variables, although I do this later as well.

The LCD was programmed with all the commands on the datasheets. I first initialised 8-bit mode, then selected 4-bit mode to reduce the amount of ports required on my breadboard. After this was initialized, I set up the other options using the other functions. I turned on the display, turned off cursor and blink, set the LCD to two line mode, set the font, and shift mode. To write to the LCD I changed the output values of PortB[3:0] and toggled the enable pin. The values I chose to write were either from the instruction set or were based on the pattern table supplied. After writing any command, I set a delay of 5ms (greater than the maximum time given for an instruction to complete) to ensure the command was processed properly.

The next part of the project that I decided to work on was the 4x4 keypad. I started by setting up the ports and defining them in my asm file so that I had Col0, Col1, Col2, Col3, Row0, Row1, Row2, Row3 instead of PortA 0-7, which made it much easier to read and work with. I set the rows to be inputs, and columns to be outputs. I wrote my code such that one column was set at any one time, and the PIC microcontroller read if one of the rows was also high. This would mean that one of the switches in that row was pressed, and hence the button could be found by looking at the row and column. If the button wasn't pressed, the row would read 0, and just loop through the check_keypad function. Function calls were assigned to these button presses to process commands.

To initialize the Si4703, I first enabled the I2C bus using the SSPCON and SSPSTAT registers. This was to set it to master mode and initialize the timing (which was determined from the values in the datasheet). Writing to the registers involved sending the start condition, and sending the address B'00100000' to the Si4703. I started by overwriting the registers up to 07h with their reset values as it meant I could get away with not reading their values and rewriting them. I did this for all registers for 02h to 06h by writing the values I wanted into SSPBUF and letting the PIC transmit them. For 07h, I enabled the crystal oscillator to establish the clock. This was done by setting the 07h register to B'10000001 00000000'. I sent the stop condition and made sure the command processed. A 500ms delay after that stabilized the clock.

The start condition and address were sent again and the chip was enabled by setting the ENABLE bit to high in the 02h register. Another stop and delay stabilized the voltages. After this, I was able to start again and write to 02h to 05h with the settings I wanted. This included turning off mute, setting the volume to 3, and setting the band and spacing. At this point the Si4703 had been initialised and was playing music.

My A button was set to turn the screen off and on (an extra feature that I included). I used another variable called screenReg to keep track of what the screen was doing. I used the screenReg variable to toggle the specific bit I needed to turn the screen on/off. This was followed by a delay, as well as complementing the screenReg to indicate that the screen had been toggled.

My B and C buttons were used for volume up and down (another extra feature). I used the variable volumeReg to keep track of the volume. This value was incremented or decremented and written to the 05h register to change the volume. When there was any overflow, the volume would stay at the max or min and display “VolAtMax” or “VolAtMin” respectively. When the volume wasn’t at the max or min, it would display “vol+” or “vol-“ to let the user know what was happening.

The D button was set to mute (another function I added). I used another variable muteReg to keep track of if mute was on or not (in order to toggle it). In a similar fashion to the other functions, I wrote different values to 02h depending on whether I wanted to turn mute on or off. If mute was on, the word “Mute” would be displayed on the screen. Otherwise the station frequency would be displayed.

My preset stations use the formula Freq = 0.2 * channel + 87.5. I had nine of them and they were each set to different frequencies.
I calculated the binary values to write by doing the calculation in decimal then converting to binary. I then wrote it to the CHAN[9:0] register and enabled tune. Following this, I provided a delay and set the tune bit to low to stop the tune operation. This delay was chosen with respect to the datasheet, which gave a maximum seek/tune time of 60ms.

Following this, the registers could be read in order to display the station on the LCD. This is what I would consider to be the most difficult and complex step. I began by sending the start condition and writing to the address B’00100001’ which includes the read bit (1). The system would wait for an acknowledge from the Si4703 and is then ready to begin reading. Once the RCEN bit is enabled, the registers are read from Si4703, starting with 0Ah. This is read into SSPBUF and must be written from SSPBUF so that the next register can be loaded. RCEN and ACKEN are cleared after each read so it can be used to show that the read has occurred. This is what is happening in my CheckProg function.

The system then sends an acknowledge back to the Si4703, and checks whether it is ready for the next register. This happens until it reaches 0Bh, which is where the READCHAN[9:0] register is. Reading this gives the current channel. An interesting technique where the channel to converted to frequency and split into digits is employed here (from Freq = 0.2 * channel + 87.5). I save READCHAN into my counter and decrement it until it has reached zero. The first digit is preloaded with 5, the second is preloaded with 7, and the third with 8. This is because I needed to add 875 from the equation. With every decrement, I increment the first digit twice (because you have to multiply by 2 from the frequency equation) until it reaches 10. I then send it to 0 and increment the second digit (overflow). By the end of the process, the digits are sorted and can be written to the LCD. The pattern table for the LCD is in ascii, so adding 48 to the binary value means it can be easily written to the LCD. This is sent to PORTB[3:0] in groups of 4 bits. The highest 4 bits are sent first, and so I needed to rotate the bits to get them in the correct order. These were sent the same as the other LCD instructions.

Seek up/down worked in the same manner except using the bits in 02h relating to seek direction and seek enable. Throughout my code, a number of bits were used to check for errors and when the systems were busy. These included SSPIF, WCOL, SSPOV,

BF, and whether RCEN and ACKEN were still set.

PIC16F886 Assembly Code
See end of document for complete .asm file.

Test Results
The way that I tested my project through each stage of implementation was by making sure that the correct things were happening, and that they were happening as expected. I started by testing my keypad since that was the first part I started on. I tested it by pressing down one key at a time and observing the program counter in MPLAB. When I held down the one button, would it run the OnePress function and only that function? I observed this result for all button presses and showed that the keypad was interfaced correctly. When nothing was pressed the code would just loop through the check_keypad function which was what I programmed it to do. However, I noticed that the functions were being run multiple times when each button was pressed. I fixed this by adding a 128ms delay.

The next part of interest was the LCD. This was probably the next hardest part to complete, though was still fairly simple. The difficult part was reading the busy flag properly, and making sure it initialized successfully. I tested this component by comparing my program with the instruction set on the datasheet. All of my code matched up with the functions provided, and did what the functions were supposed to do. I made sure that the functions were doing the correct things by changing individual bits and observing the result. For example, I turned on the cursor and observed that the cursor was shown. I then turned on the blink mode and observed that the cursor was blinking. This ensured that the code was being implemented properly. I then moved on to writing to the LCD. I tried a variety of characters and they wrote successfully. I also added a reset button to be sure that it would clear correctly. The next step was to interface the LCD with keypad. I assigned a character on the LCD to a button on the keypad, so that it would print when pressed. When the button was pressed the character did indeed appear on the LCD. With this, I was sure that those two components were functioning as expected.

The next part (and most difficult) was the FM radio module. It was difficult in the sense that it was near impossible to find where any errors were, and hence how to fix them. I started by initializing the I2C module on the PIC microcontroller. I connected it to the Si4703 module and observed on the oscilloscope that the bytes were being sent, and that the FM radio module was sending acknowledges back. At this point I knew that the radio chip was able to receive data from the I2C bus, meaning that I2C protocol was working and that the radio chip received the control address. I followed this up by powering up the crystal, enabling the chip, and initializing the registers. Each action was followed with an acknowledge which meant it was receiving the commands properly. The module starting outputting sound according to the settings I had selected which meant the commands were writing to the registers. After this I was able to implement all of my other radio functions, including the preset tuned channels, seek up/down, mute, and volume up/down. I made sure all of these functions worked and didn't halt the system. There were many hiccups throughout the process although checking all of the I2C registers for any error flags helped me to diagnose the

problems. I tested my functions by stepping through them and observing the PIC registers, and of course listening to what was being outputted. Occasionally, running the same function twice caused the system to halt. These faults were mostly due to the system not being ready for the next command. Some of the registers that I checked to diagnose these issues included the SSPIF bit in the PIR1 register, the Busy Flag in the SSPSTAT register, WCOL (write collision) and SSPOV (buffer overflow) in the SSPCON register. I also knew that receive mode would turn off after a successful receive, and that the acknowledge send bit would move low in the same situation. I used these to figure out whether the receive operation had updated SSPBUF and if not, use this to debug. The majority of errors were found by noticing what worked and what wasn't working. When any unusual behavior was encountered, I worked backwards and looked at the register values to see if something was/wasn't updating. Commenting out lines of code to deduce where the problem was also helped. Occasionally the W register wasn't updating. For these issues, I had to look at my code and find any spelling errors in then instructions, or mismatched bits (ie. bsf instead of bcf). When all the individual functions were complete, I assigned them to buttons on the keypad and ran them successively to make sure they worked together. To interface this with the LCD, I performed the same process. Making sure all variables and registers were being updated involved stepping through the code and reading the values. When something went wrong, I tried to establish what I knew was working correctly, and where the problem was. Using the watch function in MPLAB, I could pinpoint where the error was and assess it accordingly.

Discussion

Overall, the FM radio that I built worked excellently and was a success. It functioned as I programmed it to, and it fulfilled all of my design criteria. However, the model could have been improved in a number of ways. The wire placement could have been vastly simplified (since my layout was very messy). This would have aided hardware based problems as they would be easier to identify and fix. This would also have resulted in a more compact form which would be important in a commercial radio, but would also prevent damage when it was being transported. A voltage regulator could also be used to step 5 volts down to 3.3 volt, which cuts out the need for two different voltage inputs. Other improvements are related to the program, specifically that regarding some of the delays and unnecessary code. My code features a number of delay loops which exceed the amount of time required for operations to complete. This was to ensure that they were completed. The downside to this is that the program becomes inefficient and has higher time costs. Checking all of the interrupt flags and operational bits in the registers would tell us when each operation is complete, and hence allow us to run the next command sooner to maximize time efficiency.
This inefficiency occurs frequently when running the seek and tune functions. The datasheet shows that these commands take at most 60ms, although my program calls a 128ms delay. Reading the registers and checking the seek/tune complete bit would mean the next command can be issued and no time would be wasted on a delay, though this would require more code, be of greater complexity, and require more of a time commitment.

Although I didn't save the Si4703 register values anywhere and modify them in the PIC microcontrollers memory, this could be done. The updated values could then be

written back to the Si4703 to update the registers. Since this was a fairly simple project, and significant amounts of complex data did not need to be updated in the registers, I decided to just overwrite the register values with what I knew they would be. My program always returns to the same check_keypad loop, which is a predictable state, and as such, I know what the FM radio module registers will be set to. When each function is called, these can be overwritten to perform a specific function, as only a few bits needed to be changed. If the registers had to be rewritten constantly, with unpredictable values specific to the inputs received, it would be far easier to write the values to some sort of array and modify then, then save this array to the Si4703 registers. This would also make error checking much easier, but requires more memory and time as all the registers have to be read.

Many of my functions could be simplified into smaller functions to reduce the length of code (since a lot of code is reused). The sections of code where I write data to the LCD are very inefficient, as I am just changing the individual bits of PortB 0-3. It would be better just to write the value I need to the working register and move this to PortB, shifting it four places right if required (since I used a 4-bit interface and PortB 0-3).

The limitations of my project were mainly based on hardware and time. The design was quite large and bulky, and would not be convenient to carry around, especially considering that some system of batteries would need to be implemented to make it portable. The code was also fairly inefficient in that there were some pointless checks. The delays were also excessive and at times redundant. Given more time I could choose to optimize my code to make it more efficient. From a functional point of view, my design had no limitations as everything worked perfectly. The only negatives are that it receives channels of poor quality. Changing the SeekThreshold RSSI in the Si4703 registers to a different value would filter out some of the poorer quality signals (though I tried this and it didn't work). Changing the impulse threshold and SNR threshold would also fix this. Changing the initialisation sequence so that the radio starts up on a channel, instead of starting on static and then jumping to a station would also be preferable.

If given more time, it could be spent implementing more functions. Some interesting ideas for functions could include holding down a key to set the current station as one of the preset stations. Another idea is entering the station you want on the keypad (ie. pressing '9' '6' '9' for 96.9). These ideas would take a fair amount of time to implement but would be possible. It would also be interesting to display the RDS information (station name, current song, etc) on the LCD. Though this would require a significant time commitment as well.

Everything that I set out to get working from the design criteria worked in the end. As such, I can now call my project a fully functioning radio. All components worked together. No parts of my project were not working by the due date.