|  |
| --- |
| IMG_0733.JPG |
| KEYtar |
| Final Report |
|  |
| **Nick Daegele**  **Jake Darnell**  **Nik Kleber**  **Shawn Steuer**  **Zach Stauder** |
| **5/9/2013** |

|  |
| --- |
|  |

***Table of Contents***

|  |  |
| --- | --- |
| **1 Introduction** | **………………………. 3** |
| **2 Detailed System Requirements** | **………………………. 5** |
| **3 Detailed Project Description** | **………………………. 6** |
| 3.1 System Theory of Operation | **………………………. 6** |
| 3.2 System Block Diagram | **………………………. 7** |
| 3.3 Power and Battery Unit | **………………………. 8** |
| 3.4 Sound Related Inputs | **………………………. 10** |
| 3.5 Key Select | **………………………. 13** |
| 3.6 User Interface Display | **………………………. 17** |
| 3.7 Audio Output | **………………………. 20** |
| 3.8 Interfaces | **………………………. 24** |
| **4 System Integration Testing** | **………………………. 25** |
| **5 User Manual / Installation Manual** | **………………………. 28** |
| **6 To-Market Design Changes** | **………………………. 30** |
| **7 Conclusions** | **………………………. 31** |
| **8 Appendices** | **………………………. 32** |

**1 Introduction**

# Problem Description and High Level Solution

Playing a musical instrument such as a guitar is an activity which is very beneficial to the user for a number of reasons. It is an activity which can greatly help alleviate stress, and when people play instruments in groups, it is an excellent social activity as well. However, a large number of people do not know how to play a musical instrument, with less than 6.5% of Americans knowing how to play the guitar. These people are therefore excluded any time a group of their friends decide to have a jam session. This can create a great problem as it can cause those who are not musically-inclined to feel left out and to miss out on the fun which their friends are having.

Currently, if one decides that they want to start participating in these jam sessions, it is nearly impossible to do so without either extensive training or completely disrupting the music. Due to the wide range of keys which can be played in, it is inevitable that someone who is not trained in music and simply playing around on a guitar would often play notes that are out of key. This poses a great problem for that person who does not have training in a musical instrument but still wants to reap the benefits of jam sessions with friends without being a nuisance and disrupting the music.

In order to help those who are not musically inclined to still be able to play in jam sessions with friends, we are proposing a KEYtar. The KEYtar would be a very intuitive instrument which is shaped like a guitar and has two sets of buttons. One set of buttons would be a sort of keyboard positioned on the body of the instrument which would play individual notes. The other set of buttons would be positioned on the neck of the KEYtar and would be used for playing the chords within a certain musical key. In order to optimize the user experience, we are proposing the development of 3 major functionally modes of the KEYtar: **Standard**, **Assist**, and **Auto-Assist**.

**Standard Mode** is designed for the user who wants to use the KEYtar but already has at least a fair amount of musical experience. In this mode, the keyboard buttons would function in the same manner as a normal keyboard. The user would then be able to manually select the key he or she wants to play in, and the buttons on the neck would provide an easy and efficient way to play the chords in this key.

**Assist Mode** is the ideal mode for the user who is largely lacking in musical training. Like in Standard Mode, the user would manually select the key to play in and the neck buttons would play only the chords in this key. However, unlike Standard Mode, Assist Mode would also change the functionality of the keyboard on the body to only play notes within the set key. Rather than being middle C, the first key would become the root note of the key that the user sets. This allows someone without musical training to play notes on the KEYtar in any way that they like without playing out of key, leading to a much more aesthetically pleasing sound.

**Auto-Assist Mode** is designed for the user who lacks musical training but who wants to play the KEYtar in a jam session with friends. This mode functions exactly like Assist Mode except, instead of being set manually, the key in which the KEYtar plays would be determined automatically by analyzing the notes being played by the other instruments in the room. When the “listen” button is pressed, the KEYtar takes input from an on-board microphone, analyzes its frequency components to determine the key that the signal is in, and sets the KEYtar to that key. This would allow a user with virtually no concept of music to join in a jam session without having to worry about what key they are playing in.

**Expectations Met**

Our design met our expectations very well in most areas that we continued to target as design priorities. However, there were a few design elements for which we were forced to abandon progress due to time constraints toward the end of the project timeline.

Starting with the physical construction of the KEYtar, our design met our requirement of being appealing, safe, and easy to use. The KEYtar was successfully carved out of wood in the shape of a shark, painted blue-grey and white, and routed so as to store all electronics on-board in a concise and efficient fashion. After construction, the KEYtar is sleek and very attractive to look at while also being light-weight and organized in such a matter that makes operation of the KEYtar simple for the user. For example, the key buttons are placed strategically next to the right hand while the chord buttons are on the neck of the KEYtar, close to the left hand. This configuration resembles that of a real piano on which the user mainly plays chords with the left hand and melodies with the right hand. In terms of being safe, the KEYtar exhibits no characteristics which are in any way harmful to the user.

Turning to the operation of the KEYtar and its functionality, we met most of the design requirements that we set at the beginning. At the most basic level, the KEYtar successfully outputs musical signals both through an on-board speaker and through a headphone jack. Furthermore, these music signals are correctly determined by the buttons (keys and chords) which the user selects at any given time. We were very satisfied with accuracy of the output signals in terms of both frequency and power when one or several notes were being played. The button debouncing strategy that we employed also proved to be very successful as the microcontroller responded both accurately and quickly to all buttons being pressed. The effects buttons met our expectations to the extent that the selection of one of them would turn off the others and alter the output signal. However, we could not program as many or as intricate as effects as we would have liked (reverb, separate instruments, etc.) due to time constraints and the absence of an external memory chip. Even though we did not have the external memory chip to store wavetables, the real-time creation of music signals to output worked out fine for basic ramp waves without complicated sound effects. Additionally, the different modes of operation (other than automatic key sensing) worked to our liking. When in standard mode, the key select knob accurately set the chord buttons to the current key, and the sixteen segment LED accurately displayed the key selected by the user. When in assist mode, the key select knob accurately set both the chord buttons and the piano keys to the user-selected key. Rather than the first piano key being C, it gets set to the root note of the key selected by the user. In this way, our mode selection design met our requirements as we expected.

The most significant design specification that did not meet our expectations was the rechargeable battery unit. Our design plan was to include rechargeable batteries and a charging circuit on the KEYtar along with the ability to plug into the wall for instantaneous power. As the project deadline came closer, we decided not to make the rechargeable batteries a priority as there were aspects of the project that were much more crucial to create a working product. We ended up designing a power board to accept power from a wall jack, and rather than having a battery charging circuit on-board, we included a terminal block to accept a power input from an external battery unit for possible use in the future.

All in all, the KEYtar met our expectations very well both in terms of its physical and appearance and in terms of its functional operation. Though there were some design subsystems that we could not fully implement in the finished product, all of the subsystems that we ultimately included performed very much to our liking, and to the liking of everyone that had a chance to play the KEYtar on demo day.

**2 Detailed System Requirements**

|  |
| --- |
| Subsystem and Interface Requirements |
| ***Power and Battery Unit*** |
| **General** Must be able to power KEYtar directly from U.S. wall outlet or charged batteries |
| **Size** Batteries must fit unobtrusively inside the KEYtar |
| **Weight** Must weigh less than 2 pounds in total |
| **Power** Must output constant adequate power for the system  Power estimate is about 1.2 Amps |
| **Microcontroller** Must sense if the device is charging  **Software** Must sense the voltage of the batteries and adjust power cycle as necessary |
| **Batteries** Must be rechargeable, fully recharged in less than an hour  Must have a battery life of at least 30 minutes  Circuitry must be able to protect batteries |
| **Plug-In Charger** Must have a sufficiently long chord  Must be able to charge batteries and directly power KEYtar |
| ***Sound Related Inputs*** |
| **General** Must be able to accurately send signals to the microcontroller corresponding to the sound that the user is attempting play |
| **Size** Chord buttons must be approximately 1.5 finger widths wide and  one inch tall, rising slightly above the KEYtar surface  Note keys should be the approximate width of a standard piano key  while being between 3 and 5 inches tall  Sound manipulation buttons should be slightly smaller than a dime |
| **Power** Must draw less than 200mW |
| **Microcontroller** Must accurately detect the signals from sound inputs  **Software** Digital debouncing filters must prevent microcontroller from incorrectly detecting notes and switching on and off because of bouncing |
| ***Key Select*** |
| **General** Must be able to designate a key for notes and chords to be played  either through user selection or automation from ambient sound |
| **Microcontroller** Must be able to execute an FFT of frequencies and use a lookup  **Software** table to determine the key  Must map piano keys according to selected key |
| **Mode Switch** Must have three states (Standard, Assist, Auto-Assist)  Must be less than one inch long |
| **Key Select Knob** Must be larger than a dime and smaller than a quarter  Must be an incremental encoder rather than absolute encoder  Must be able to increment up and down through keys |
| **Listen Button** Must be size of a fingertip (1cm diameter circle)  Button will be pressed once, activating microphone and disabling  audio output so that microcontroller can determine key |
| **Microphone** Must pick up sound of standard noise levels  Must be surface-mounted, flush with KEYtar  Must have surface area less than 4 cm squared |
| ***User Interface Display*** |
| **General** Must accurately convey to the user the current KEYtar settings |
| **Size** LEDs: standard bulb LEDs  7 segment: standard 7 segment display (1.5” x 2.5” x .15”) |
| **Power** Must consume less than 65 mW total |
| **LEDs** There must be LEDs indicating: power on or off; low battery; battery charging; battery fully charged; key identified; sound manipulation identifications |
| **7 Segment** Must output the key selected  Period represents sharp symbol |
| **Microcontroller** Must have a lookup table for 7 segment display  **Software** Must provide high/low output for LEDs  Must only light up one LED at a time, the one pushed most recently |
| ***Audio Output*** |
| **General** Must be able to output sound to speaker or output jack at a user-  controlled volume  Speaker must turn off when output jack is connected  Output jack is a quarter inch TRS  Speaker must be mounted to KEYtar |
| **Size** Speaker must be a big as possible within remaining available space  Volume control knob must be ¾” in diameter and rise above  KEYtar surface by ½”  Output jack is ¼” |
| **Weight** Speaker should weigh less than 1 lb |
| **Power** Speaker must consume less than 50 W |
| **Microcontroller** Must have digital to analog converter to output real-time audio  **Software** signal from digital data  Must be able to add separate frequency components  Must be able to update analog value that is being output at twice the maximum frequency component of the audio signal  (≥ ~40kHz) |

**3 Detailed Project Description**

**3.1 System Theory of Operation**

The major components of the overall systems are the Power and Battery Unit, Key Select, Sound Related Inputs, User Interface Display, and Audio Output systems. All of these components interface with the two microcontrollers for the operation of the KEYtar.

The Power and Battery Unit will supply a sufficient amount of power to the KEYtar. The Power Unit uses a wall wart to accept a DC voltage from a power block. Two voltage regulators provide a stable voltage of 5V and 3.3V to the system. Rechargeable batteries may be connected to the unit to power the system instead of the DC source.

The Key Select system will take inputs from the Mode and Key Select blocks in order to determine the key in which the KEYtar should play. In Standard Mode and Assist Mode, the user sets the key select knob to specify the key. The hardware for the listen button and microphone are functioning for auto-assist mode, but the auto-assist mode is yet to be implemented.

The Sound Related Inputs include the individual notes that are being played, the chord buttons, and various other sound effects. The output pitches will be determined by the microcontroller based upon the individual buttons/keys that the user presses. The chord buttons will correspond to the chords within the specified key. The sound effects include two types of sound distortions that make the sound ethereal.

The User Interface Display consists of a sixteen segment display and various LEDs. The sixteen segment display will be controlled by the PIC24 microcontroller and will display the key in which the KEYtar is currently operating. The LEDs are positioned on the KEYtar near user inputs to indicate user selections. The LEDs will also be controlled by the microcontroller.

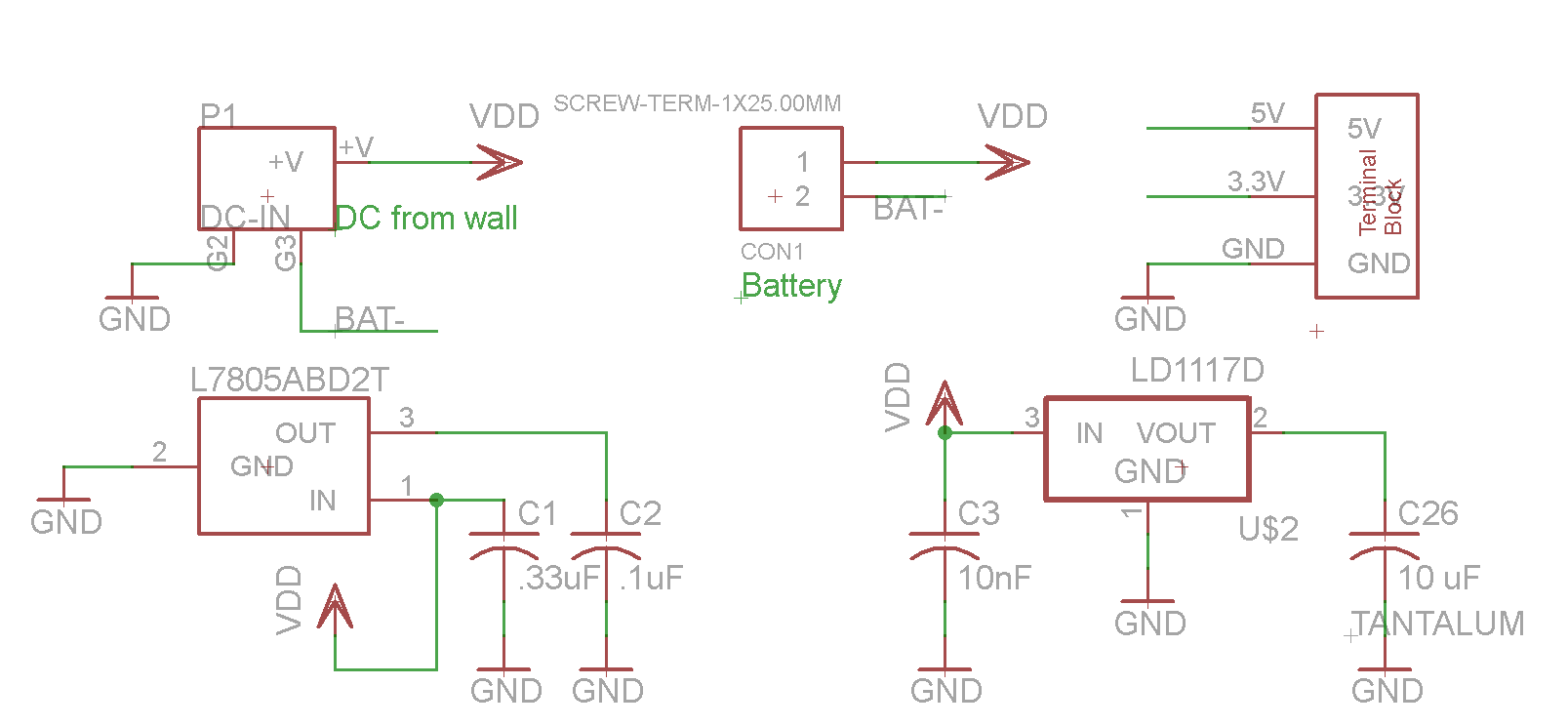
The Audio Output system consists of an output jack, a speaker, an amplifier, and a volume control knob. The audio output is directed either through the amplifier to the on-board speaker or through the output jack. The output jack does not have volume control, but the on-board amplifier does. The user accesses the volume control through a knob.

|  |
| --- |
| **3.2 System Block Diagram** |
|  |
| **Figure 3.2.1. The KEYtar system block diagram** |

**3.3 Power and Battery Unit**

The subsystem requirements for the Power and Battery Unit are not too complicated. Originally, the requirements for this unit were intended so that we would design our own charging circuit. Due to time constraints, we decided to have an external battery and charger as well as a place to plug in a DC voltage. Overall, this system met all its specifications, though. It was able to power the system. It was small and unobtrusive. And the batteries are rechargeable and can power the system as well. The only shortcoming was that the batteries were not mounted in the KEYtar due to time constraints.

The schematic for the power board is shown in Figure 3.3.1.



**Figure 3.3.1. Power Board Schematic**

The key design decisions made for this subsystem were the voltage regulators. We needed two voltage regulators. One would output 5 volts to power the audio output. The other would output 3.3 volts to power the rest of the microcontrollers, the LEDs and the remaining electrical components. Our original idea was to set up the regulators in series. This way the input to the 3.3-volt regulator wouldn’t be too high, and the 5-volt regulator would deal all of the current. Based on this decision, we needed a regulator that would source more than one amp. We calculated our system to source just over an amp in the worst-case scenario. Therefore we chose the L7805ABD2T regulator because it could output 1.5 amps, which was a decent safety factor. The LD1117D was already available to us, and would source enough current for the 3.3 volts. That was an easy decision to make. The 5-volt regulator takes an input from 5-35 volts and the 3.3-volt regulator takes inputs up to 15 volts. Therefore, we decided that a 12-volt input would be sufficient.

After deciding on our parts, we came to the conclusion that it might be better to put the regulators in parallel. This way, one regulator doesn’t have to source all the current. They could share it about equally, since the audio output would source about as much current as the rest of the system. The capacitors chosen were indicated in the data sheets for each regulator.

Another design decision we made was to make it so that the batteries would power the system whenever the wall jack wasn’t inserted. Then when it was plugged in, the batteries became disconnected. This was accomplished by using the internal switch in the DC barrel connector. When something is plugged into the wall, the batteries are no longer connected to ground, breaking that circuit so that it doesn’t interfere with power coming from the wall.

Finally we chose terminal blocks to transfer power. The screw-in feature was attractive because it seemed that it would connect wires to the board without the risk of them being yanked out. One terminal block delivers power to the “*Microcontroller Board*”. The other terminal block receives the power from an external battery.

In order to test the Power Board, we first checked to see if there were any connections to power and ground. When it was open, we connected 12 volts to the input, and tested the output voltages to make sure they were 5 and 3.3 volts.

**3.4 Sound Related Inputs Subsystem**

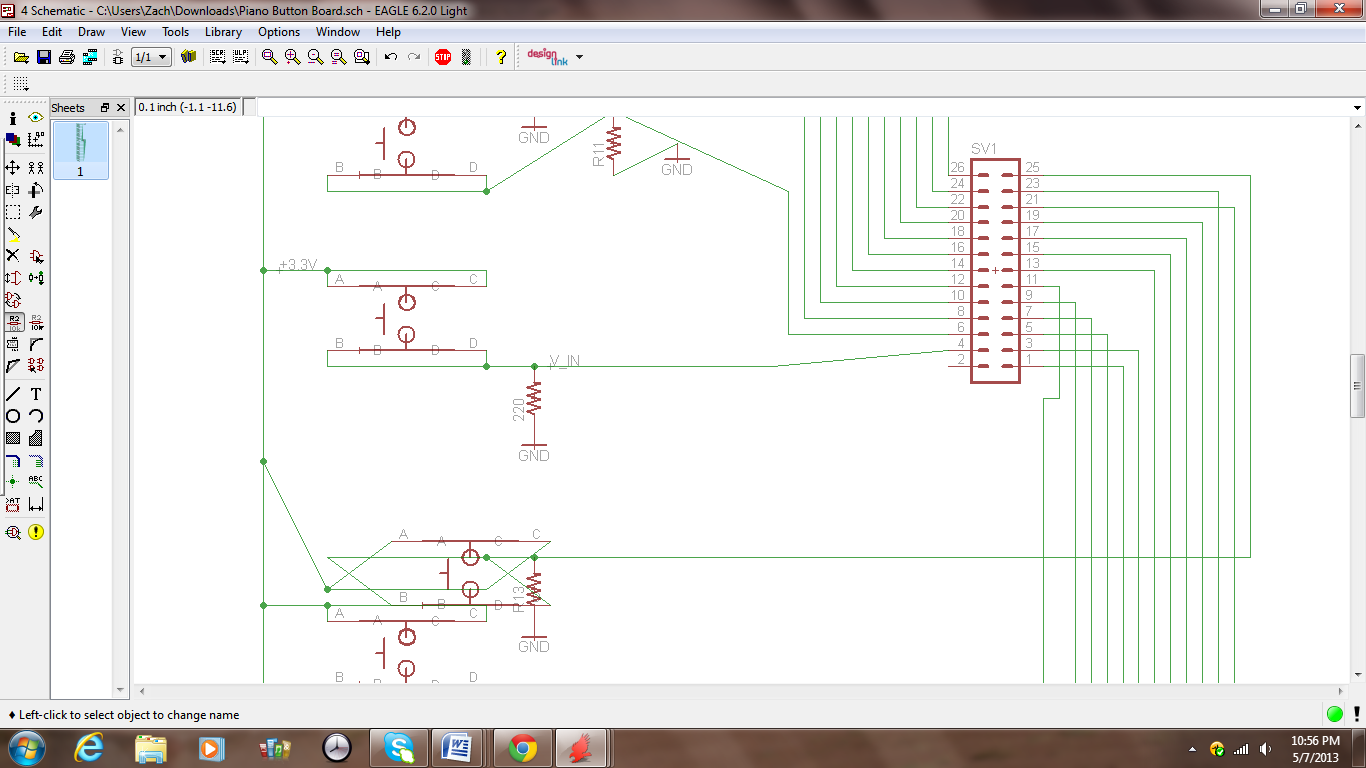
*3.4.1 Subsystem Overview and Requirements*

The sound related inputs subsystem is the system which takes the user input of which keys and/or chord are being played, maps these inputs to the corresponding frequencies, and then sends these frequencies to be played to the dsPIC so that these frequencies are actually output. This subsystem’s inputs consist of the 25 piano keys, the 7 chord buttons, and the 6 effects buttons. Each of these buttons serves as an input to the PIC24 microcontroller, which calculates the corresponding frequency associated with each button and sends the resulting frequencies to the dsPIC using I2C.

|  |
| --- |
| ***Sound Related Inputs Requirements*** |
| **General** Must be able to accurately send signals to the microcontroller corresponding the sound that the user is attempting play |
| **Size** Chord buttons must be approximately 1.5 finger widths wide and  one inch tall, rising slightly above the KEYtar surface  Note keys should be the approximate width of a standard piano key  while being between 3 and 5 inches tall  Sound manipulation buttons should be slightly smaller than a dime |
| **Power** Must draw less than 200mW |
| **Microcontroller** Must accurately detect the signals from sound inputs  **Software** |

*3.4.2 Subsystem Hardware*

The hardware component of this section included the 25 piano keys, the 7 chord buttons, the 6 effects buttons, and the PIC24 microcontroller. Each of the buttons was implemented as a pull-up input to the microcontroller. Thus, when each button was not pressed, the corresponding input to the microcontroller was grounded through a 220Ω resistor for 0 digital input. When each button was pressed, the input to the microcontroller was then pulled up to VDD, which was 3.3V. A schematic showing the connection of a button to the microcontroller input is shown in Figure 3.4.1.



**Figure 3.4.1 Schematic: Button Connection to Microcontroller**

For these connections, it was determined that a 220Ω resistor would be an acceptable value as it was desired to limit the possible current into the microcontroller to 20mA while also limiting the power dissipated when the buttons were pressed.

The physical buttons used were chosen based on the requirements determined at the initiation of the project. For the chord buttons, a simple off-mom square push-button was used. The manufacturer part number for this button is B3J-1100. This button was determined to be the desired width and provided the desired amount of resistance when pressed. For the effect buttons, an off-mom push button with an LED was chosen with manufacturer part number CFPB-1CC-5G9W. This button was in accordance with the requirements in terms of size, and was chosen primarily due to its inclusion of an LED, as described in the User Interface Display subsystem. Finally, for the piano key buttons, an off-mom push button was chosen based on its resistance and travel when pressed. Using a pine wood, the 26 piano keys were cut out and mounted on top of these buttons to form piano keys. These wooden piano keys were connected together by a rod which allowed them a small degree of rotation, in the same manner that piano keys can rotate slightly when pressed.

*3.4.3 Subsystem Software*

In order for the microcontroller to properly handle the user inputs, it was first necessary for the PIC24 to be able accurately determine when a button had been pressed. Thus, it was necessary to have software which would “debounce” the buttons to account for occasional bouncing at the inputs to the microcontroller. This was implemented using a Schmidt Trigger in which 25% of the current value of the input was added to 75% of the previous value calculated by the microcontroller. If the new value was found to greater than 0.5 then the button was determined to be pressed, whereas otherwise it was determined not to be pressed. In this manner, if the button was pressed the input would rise high, while if the button was released it would take a few cycles through the software for the input to go low. Thus, slight fluctuations at the input would not affect the state of the button being used by the microcontroller. A portion of the code used to debounce a single button is shown below:

**temp = (oldvalue >> 2);** // divide oldvalue by 4

**oldvalue = oldvalue - temp;** // create 0.75\*oldvalue by subtraction

**if(EF1\_B==buttonPress){oldvalue = oldvalue +0x3F;}** // 0x3F is about 0.25 of a uint8

**if((oldvalue[number] > 0xF0)&&(flag[number]==0)){flag[number]=1; output[number]=1;}**

**if((oldvalue[number] < 0x0F)&&(flag[number]==1)){flag[number]=0; output[number]=0;}**

After the debouncing, the main step in software was to take the current key selected for the KEYtar and to map the piano and chord button inputs to the correct frequencies. Given the 12 keys that the KEYtar can play in and the 25 piano button inputs, it was determined that there were 36 possible frequencies that could be played at any point in time, and thus a variable was created with enough bits to contain each of these frequencies. If the key is C, the piano buttons are simply mapped directly to this variable “playThis”, whereas if the key is something other than C a simple bit-shift occurs in this variable to shift the frequencies being played corresponding to the key. In a similar matter, the frequencies of any chord being played were added to this same variable and then bit-shifted according to the corresponding key.

For the 6 effects buttons, after debouncing it is determined if a new effect button has been pushed. Only one effect is allowed to be turned on at a time, so if one button is pushed, the state of each of the other effects buttons is turned “off”. This is done by using the microcontroller output ports to turn off the corresponding LEDs, as described in more detail in the User Interface Display subsystem section. The bit corresponding to which effect button is in the on-state is included with the *unsigned long long* variable for which frequencies are to be played and is sent from the PIC 24 to the dsPIC33 using I2C. Overall, the microcontroller was constantly polling the state of each of these user inputs within its main file. This given the operating speed of the microcontroller, it was determined that the state of each input would be checked approximately every 1 ms.

*3.4.4 Subsystem Interfaces*

In order to connect the user inputs to the PIC24 with the dsPIC33 microcontroller which is to output the sound, an I2C interface was used. This interface will be described in more detail in the *Interface* section, but essentially consisted of sending 5 bytes of information from the PIC24 to the dsPIC. The first 36 bits correspond to the 36 possible frequencies which could be played, while the last four bits correspond to the state of the first four effects buttons which were used and of which only one could be on. This information was sent from the PIC24 at the end of each cycle, which corresponded to about every 1 ms due to the 100 kHz clock rate at which this microcontroller was set. Each time a byte was sent, an interrupt was generated in the dsPIC code which would cause an interrupt service routine to interpret the sent byte.

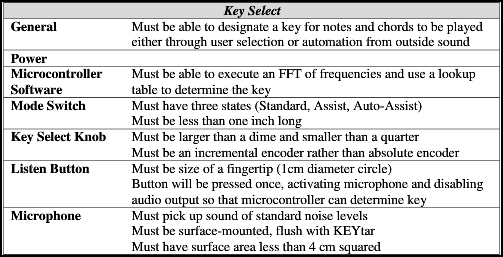
*3.4.5 Subsystem Testing*

In testing this subsystem, the first feature tested was the debouncing of the buttons. This was done by simply connecting a few of the buttons to a through-hole prototype board and then connecting the inputs to a microcontroller on the kit board. Once it was determined that the debouncing was working correctly, the rest of the code associated with the user inputs was added and implemented.

Before the software was implemented on the PIC24, the buttons and corresponding resistors were mounted to the board and a digital multimeter was used to measure the voltage input when the button was and was not being pressed. In this manner, it could be determined that the proper values were being input to the microcontroller.

The frequency mapping and effect button selection portions of the software implemented after the audio output subsystem was already functioning as desired. Thus, in order to test these algorithms, the five bytes of data containing this information simply had to be sent to the dsPIC using I2C. When these subsystems were connected, it was determined that pressing the piano keys and chord buttons resulted in the correct audio output frequencies, and thus this portion of the subsystem was determined to be functioning properly.

**3.5 Key Select Subsystem**

Below are the subsystem requirements for the Key Select subsystem

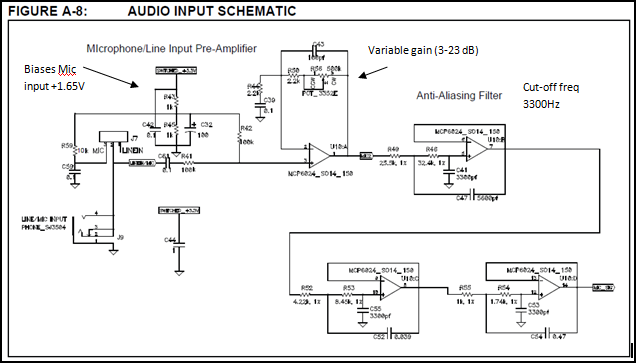
**Table 3.5.1. Requirements for Key Select subsystem.**

In regards to meeting the subsystem requirements, the Key Select subsystem that we built did not meet all of its original requirements. It does not automatically detect the key. The only Key detection algorithm that we got working was on Matlab. Other than that, the mode switch works, the key select knob sort of works, the listen button is not used, and the microphone is not used, but based on our tests, the microphone, pre-amp, and filter send the correct signal to the A/D pin of the DSP chip.

There are four parts to the Key Select subsystem. They are Microphone/pre-amp/filter, quadrature encoder (key select knob), key and chord button mapping software, and automatic key sensing software. Below are the notes of each component in the Key Select subsystem.

**Microphone, Pre-amp, and Filter**

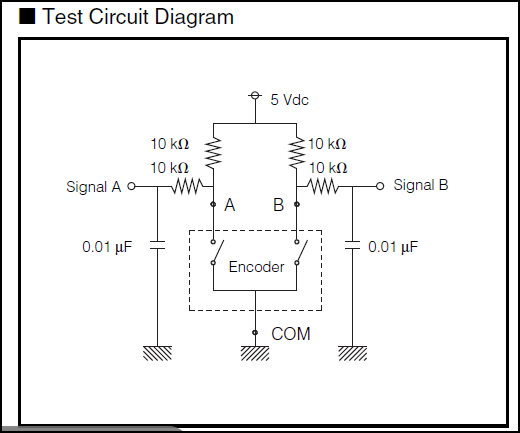
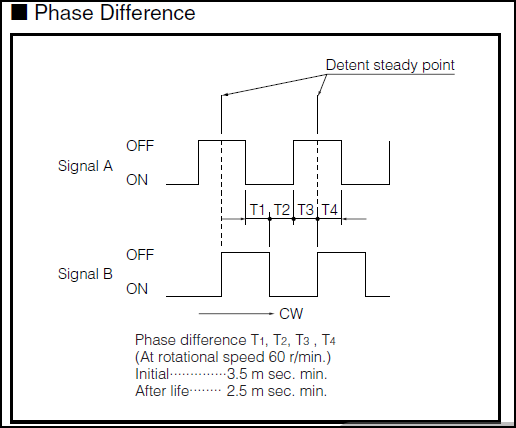
The microphone, pre-amp, and filter were essentially designed based on the pre-amp and filter used on the Microchip DSP development board. The only difference is that we used a different microphone that we mounted directly to the circuit board. We chose to use this design because we were able to test it using the development board, and it worked well enough for our purposes. Since the testing was sufficient, we make our lives easy and did not try to make a new design. The basic design is a 3 op-amp filter and a 1 op-amp pre-amplifier between the microphone and the A/D converter pin. See below for the schematic and notes about the system.

**Figure 3.5.1. The microphone, pre-amp, and filter on which the design was based.**

* Pre-amp on dsPIC uses potentiometer to tune the gain of the op-amp via feedback (changes gain of mic/line input) from 3 dB to 23 dB
* Anti-aliasing filter uses 3 op-amps with decreasing resistor values to filter out high frequencies.
* 6th order Sallen-Key low pass filter with a cutoff frequency of 3300 Hz.
* If the input to the amplifier is a condenser microphone, a bias voltage provides a working supply voltage for the microphone.
* Output of amplifier is biased at 1.65 V.
* On dsPIC if MIC is selected (over line) then a bias voltage of +3.3V is applied to input
* Basically, the dsPIC uses a MCP6024 chip, which is 4 op-amps. One is used for pre-amp, and the other three are used for filtering.

**Quadrature encoder**

The quadrature encoder circuit is pretty simple. The quadrature encoder is simply made of two mechanical switches that click on and off 90 degrees out of phase. For a more in-depth explanation on the quadrature encoder, search the internet for “incremental quadrature encoder.” Since this is just two switches, the circuitry around it is almost as simple as the circuitry for the other buttons on the KEYtar, except with a low-pass filter attached. Below is a diagram of the schematic that we used, as well of a diagram of the output signals coming from the quadrature encoder as it is turned.

**Figure 3.5.2. The quadrature encoder operation.**

We chose this particular quadrature encoder because it has detents, and goes through 24 cycles per revolution which seemed to be about the appropriate amount for our application.

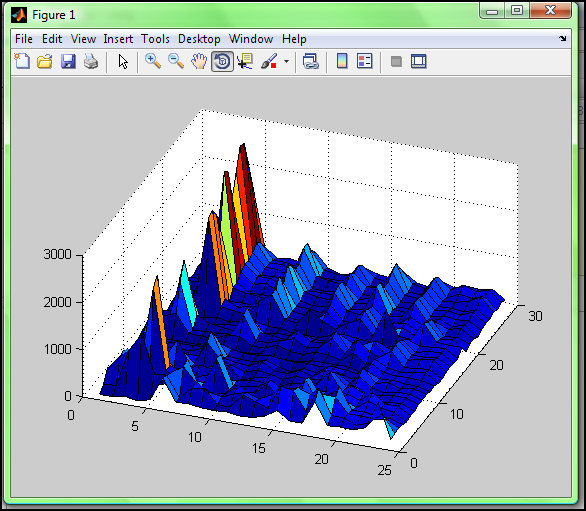
**Key and chord button mapping software**

The Key and chord button mapping software included information about what chord buttons and what piano buttons were connected to which inputs of the microcontroller. It took in the information about which notes were being played, and it shifted them up based on what key was selected. This information was then sent to the DSP chip so that the DSP chip could produce the correct output frequencies. The table of information about which notes corresponded to which pins on the microcontroller is listed below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 3.5.1. The mapping of the user interface to the pin and software label.** | | | |
| **Button** | **Physical label** | **Pin** | **Software label** |
| C1 | PIANO2 | RE8 | PIANO1 |
| C# | PIANO4 | RB5 | PIANO2 |
| D | PIANO6 | RB3 | PIANO3 |
| D# | PIANO8 | RA10 | PIANO4 |
| E | PIANO10 | RB9 | PIANO5 |
| F | PIANO12 | RB11 | PIANO6 |
| F# | PIANO14 | RF13 | PIANO7 |
| G | PIANO16 | RB12 | PIANO8 |
| G# | PIANO18 | RB14 | PIANO9 |
| A | PIANO20 | RD14 | PIANO10 |
| A# | PIANO22 | RF4 | PIANO11 |
| B | PIANO24 | RF3 | PIANO12 |
| C2 | PIANO1 | RA0 | PIANO13 |
| C# | PIANO5 | RB4 | PIANO14 |
| D | PIANO3 | RE9 | PIANO15 |
| D# | PIANO7 | RA9 | PIANO16 |
| E | PIANO9 | RB8 | PIANO17 |
| F | PIANO11 | RB10 | PIANO18 |
| F# | PIANO13 | RA1 | PIANO19 |
| G | PIANO23 | RF5 | PIANO20 |
| G# | PIANO25 | RF2 | PIANO21 |
| A | PIANO21 | RD15 | PIANO22 |
| A# | PIANO19 | RB15 | PIANO23 |
| B | PIANO17 | RB13 | PIANO24 |
| C3 | PIANO15 | RF12 | PIANO25 |
| 1 | CHORD7 | RG8 | CHORD1 |
| 2 | CHORD5 | RG6 | CHORD2 |
| 3 | CHORD3 | RC3 | CHORD3 |
| 4 | CHORD6 | RG7 | CHORD4 |
| 5 | CHORD4 | RC4 | CHORD5 |
| 6 | CHORD2 | RC2 | CHORD6 |
| 7 | CHORD1 | RC1 | CHORD7 |

**Automatic key sensing software**

The automatic key sensing software was not completed on the KEYtar, but a working proof-of-concept was created on Matlab. The basic software flow was as follows.

1. Create a spectrograph of FFTs with a sampling frequency of 44.1 kHz and sampling length of 0.3 seconds. Take 30 of these FFTs.
2. Eliminate noise by zeroing out signals that are below a certain threshold value.
3. Once the spectrograph is created, put all the values between specific note boundaries into 25 “buckets” corresponding to two octaves worth of notes, and the plot of the values of those buckets becomes the new frequency spectrum.
4. At this point we have a 25x30 spectrograph. (See this spectrograph in the figure below)
5. Sum this spectrograph together along the time axis, which gives a 1x25 vector and its values correspond to the presence of notes that were played during the 9 seconds of total sampling time.
6. Turn this 1x25 vector into a 1x12 vector by summing repeated notes into different octaves. Now you have a vector that basically says, “C is played this much, C# is played this much, etc.”
7. Compare this 1x12 vector to a set of 12 1x12 vectors that correspond to the expected prominence of notes in different keys. Use an R2 error method.
8. Whichever comparison results in the lowest error corresponds to the key that the music is being played in. Since this decision is made purely on which key the music is closest to, it will always pick a key. Even if the algorithm listens to only noise, it will still pick a key.

**Figure 3.5.3. The spectrograph of FFTs.**

**3.6 User Interface Display Subsystem**

*3.6.1 Subsystem Overview and Requirements*

The user interface display subsystem consists of a number of LEDs to let the user know which effect(s) is being used and to which key the KEYtar is set. This was all done on the board with the PIC24 as this microcontroller was responsible for determining the current key and the current effect(s), and then used its output pins to light up the corresponding LEDs. The requirements for this subsystem are shown below:

|  |
| --- |
| ***User Interface Display Requirements*** |
| **General** Must accurately convey to the user the current KEYtar settings |
| **Size** LEDs: standard bulb LEDs  7 segment: standard 7 segment display (1.5” x 2.5” x .15”) |
| **Power** Must consume less than 65 mW total |
| **LEDs** Must be an LED indicating: power on or off; low battery; battery charging; battery fully charged; key identified; sound manipulation identifications |

*3.6.2 Subsystem Hardware*

In order to display to the user which of the 12 major keys the KEYtar is selected to play in, a 16-segment LED display was chosen from Lite-On Inc with part number LTG-587G. This 16-segment display was chosen over the initial requirement of a 7-segment display so that capital letters representing each key could be successfully displayed, along with an extra dot to indicate if the key was a sharp. The chosen 16-segment display was found to be of sufficient size for the user to see. The schematic used to connect this 16-segment LED to the PIC24 microcontroller is shown below in Figure 3.6.1.

# 

**Figure 3.6.1 Schematic for 16-Segment LED Display**

As can be seen in this schematic, the 16 segment display was provided the positive 3.3V VDD at one pin. Each of the other pins was then connected to an output pin of the PIC 24 with a 330Ω resistor in series. Each LED segment behaved like a diode with the anode at 3.3V and the cathode controlled by the PIC24 output pins. Therefore, in order to light up any segment, the microcontroller simply has to reduce its output to 0V, causing the diode to be forward biased and current to flow. Neglecting the voltage drop across the diode itself, it is found that with the 330Ω resistor, the maximum current into the microcontroller would be 10mA, which is below the maximum value of 20mA and which would remain below the maximum 200mA total input even if all of the segments were lit, which is never the case for this program. Furthermore, the maximum power dissipated per segment was found to be about 33mW. Although this would lead to more total power dissipated than desired given the requirements, it was determined to still be a reasonable value.

As mentioned in the Sound Related Inputs subsystem section, the LEDs to indicate the states of the effects buttons were actually included in the Copal Electronics buttons used. Although the button and LED shared the same package, schematically they were completely isolated from each other and thus can be treated separately. The LED portions of the buttons were connected to the PIC24 microcontroller via a 100Ω resistor, as shown below in Figure 3.6.2.

# 

**Figure 3.6.2 – Schematic for LED Buttons**

# 

In the same manner as the 16-Segment Display, the LEDs in the button were turned on simply by setting the corresponding PIC24 output pins low and forward biasing these diodes. Given that the voltage drop across the LEDs was found experimentally to be around 2V, it was determined that the maximum current into the microcontroller would be 13mA, which is below the required limit. It was further determined that this lit up the LEDs to a sufficient level to be seen while still dissipating minimal power. Furthermore, due to the fact that software limited these buttons so that only one could be lit at any given time, power dissipation was especially minimized.

Although initially there were going to be separate LEDs included on the KEYtar to indicate whether power was applied and to indicate the state of battery charging, these LEDs were not included in the final design. The battery charging LEDs were found to be no longer applicable since this subsystem was not included in the final project. Furthermore, because one of the effects button was deemed to be on by default and since the 16-Segment Display defaulted to reading a key of “C” upon the introduction of power to the KEYtar, it was determined that there were sufficient visual displays to indicate to the user when the KEYtar was powered. Thus, a separate LED to simply indicate power was determined not to be necessary.

*3.6.3 Subsystem Software*

The software for the User Interface Display subsystem is very simple and straightforward. For the 16-Segment Display, there was simply a header file function written which would take as its input the key selected and which would set the 17 pins associated with the 16-Segment high or low so that the correct letter was displayed. In the main file for the PIC24, each time the key was changed in the Key Select subsystem, this function to set the 16-Segment was simply called and implemented. This occurred within the main cycle of the PIC24’s code file, which was executed in full approximately every 1 ms.

For the effects buttons’ LEDs, the state of the LEDs was directly related to whether a current effect was considered on or off. Therefore, the algorithm to light up each LED was intertwined with the code to determine which effect was pressed, as described in the Sound Related Inputs subsystem section. Thus, each time a button was pressed, if the LED was previously off it would then turn on and vice versa. If a new LED was turned on, there was then code to turn all of the other LEDs off by setting the output pins associated with these LEDs high. If an LED was turned off, then the first effects LED was then turned on as a default.

*3.6.4 Subsystem Interfaces*

There were no complex interfaces between the User Interface Display’s components and the microcontroller, as they were simply connected directly to output pins of the PIC24 as outlined above.

*3.6.5 Subsystem Testing*

Testing the 16-Segment display was very straight forward as this component was simply installed on the PCB with the corresponding resistors and VDD supplied. When 0V was then placed on each pin, it was determined that the corresponding LED segment lit up to a sufficient degree.

Testing the effects buttons’ LEDs proved to be a little more complex as initially a much larger resistor on the order of 1kΩ was placed in series with each LED. It was found that with such high series resistance, the LED barely lit up to a noticeable level. This was determined to be largely in part because the voltage drop across the forward biased LED was significantly higher than expected given the data sheet. Thus, the value of this resistance was gradually decreased until the LED lit up to a level that it could be seen to be on in a lit room. The current through the LED was then tested as well and it was found to still be less than 15mA, and thus small enough to be sunk by the PIC24 microcontroller.

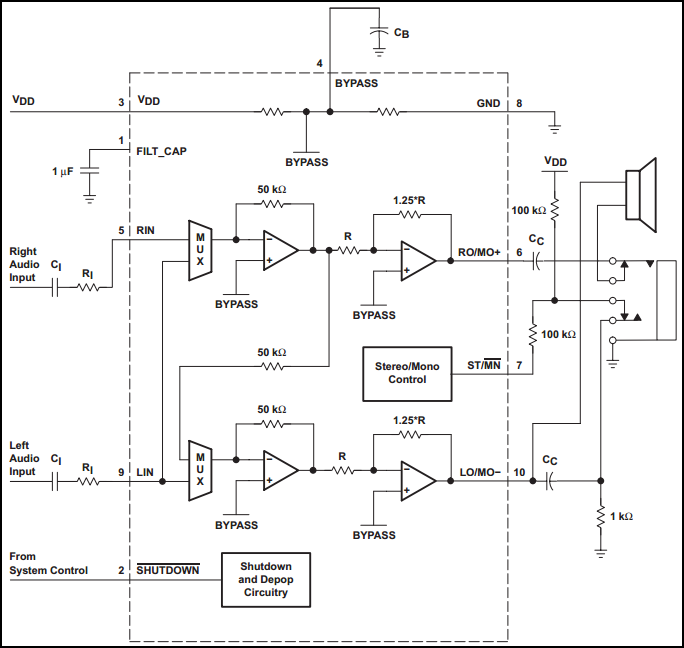
**3.7 Audio Output Subsystem**

**Audio Output Hardware**

The hardware of our audio output subsystem is designed to receive an analog signal from the digital to analog converter on-board the 33 series dsPIC and play it through either the speaker on the KEYtar or the headphone jack. The DAC outputs three lines. The positive and negative lines output opposite signals centered around the third line, which is typically held at a DC value of 1.65 V. The first stage of our circuit, which is depicted in Figure 3.7.1 functions to take the differential value out of the DAC and convert it to a single ended value that is supported by the audio amp. That single ended signal is then passed through a potentiometer, which acts as the volume control. Immediately following the potentiometer, the signal is sent through an operational amplifier whose function is to keep the voltage of the signal independent of the load connected. This, in a sense, is a buffer for the output that protects it from any unexpected problems with the load.

The signal then passes from the buffer circuit to the audio amplifier, which is a TPA0233 made by Texas Instruments, and is represented in Figure 3.7.2. The amplifier takes in two inputs and either sums them in a bridge-tied-load (BTL) configuration to drive speakers or drives two single-ended signals into a headphone jack. The two inputs are just the signal from the buffer circuit split into a fork, creating identical signals with half the current going into the left and right inputs of the audio amplifier. The inputs are passed through simple filters consisting of one capacitor and one resistor. These form a single pole high pass filter to bias the input signal to a dc level proper for amplifier operation. The gain of the amplifier is set by the input resistors only, as the feedback resistors are located in the IC. The gain is the ratio of the feedback resistor (50 kilo-ohms) over the input resistor, which we chose to be 40 kilo-ohms. The output of the TPA0233 has two channels. The output circuit allows for automatic switching between the speaker and the headphone jack. When headphones are plugged into the headphone jack, the lines from the audio amp output to the speaker are separated and connected to the headphone plug. At that point, the amplifier goes out of bridge-tied-load operation and switches to single-ended mode, as mentioned before. The speaker we selected has an impedance of four ohms and can drive up to four watts of power.

**Figure 3.7.1. Audio Pre-Amp Circuit**



**Figure 3.7.2. TPA0233 Audio Amplifier**

**Audio Output Software**

The system requirements for the audio output software were simple. It needed to take in the inputs from the *Microcontroller*, then play the corresponding frequencies. The KEYtar certainly met this requirement. We also originally wanted it to determine the key of surrounding music, but due to time constraints, we weren’t able to get the Analog to Digital Converter working.

The audio output was also limited because the memory in the *dsPIC* was limited. Because we weren’t able to communicate with the Serial Flash Memory, we were not able to store different timbres and effects, which was our original hope. Instead, the *dsPIC* stored one period of a saw tooth wave so that it could create a frequency with that waveform. We chose this because it was easy to calculate for the *dsPIC* and it didn’t occupy much memory. The software flow graph for this system is shown in Figures 3.7.3 and 3.7.4.

|  |
| --- |
|  |
| **Figure 3.7.3. I2C Interrupt Service Routine** |

|  |
| --- |
|  |
| **Figure 3.7.4. DAC Software Flow Graph** |

Figure 3.7.3 shows how the input notes are received from the *Microcontroller*. The I2C interrupt flag is set whenever the *Microcontroller* is trying to communicate with it. The notes are received and are placed in an array of length 36 of ones or zeros. Each index represents a frequency (C, D, D# etc.), and if its value is one, then that note should be played. Also in that service routine a multiplier is determined. This changes based on how many notes are being played.

This was a significant design decision for the team. We decided it was not desirable to always output the max volume from the DAC. If that happened, we would press one key and then 2 more and the volume of each note would significantly decrease. We wanted a system that would give a constant volume for each note being played. This was also difficult to achieve. If each note were given 1/36 of the possible volume, the notes would be hardly audible. Therefore, we decided to multiply the total volume by a factor, which was based on the number of notes being played. Multiplying a single frequency by 16 would yield a max volume output. Table C shows the choice of multipliers based on the sum of input notes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 3.3.1. Multiplier Values** | | | |
| **Sum of Notes** | **Multiplier Value** | **Sum of Notes** | **Multiplier Value** |
| 1 | 8 | 5 | 3 |
| 2 | 5 | 6 | 2 |
| 3 | 4 | 7 | 2 |
| 4 | 3 | 8 or more | 1 |

The DAC software flow (shown in Figure 3.7.4) starts by initializing the clock, I2C, the DAC and the counters. Choosing the clock rate was important because it determined how much faster the *dsPIC* operated compared to its DAC output. The DAC clock was set because we had a sampling rate of 44.1 KHz, and the DAC clock must be set at 256 times the sampling rate. We decided to make the *dsPIC* operate at 10 times the speed of the DAC clock, making it go through 2560 clock cycles per DAC output. We used the Fast RC oscillator to achieve this speed, and after the PLL, the final clock frequency was 56.4 MHz. The DAC was then enabled and configured using signed integers operating at one tenth the frequency of the main clock.

For each frequency, there is a counter that keeps track of where that frequency is in its period. It is incremented every time through the loop until it reaches the period of that frequency. Then the counter is set back to zero.

The software runs through all 36 frequencies to determine if it is to be played. If so, that value determined by the counter is added to the output. Then the counter is incremented. This happens for all 36 notes, and the output value is multiplied by the multiplier specified in the interrupt service routine. Finally, that final output is written to the queue to send out on the DAC. If the DAC queue is full, the software waits until it is not full. Then the whole process is repeated. If the same notes are being played, then the same counters are incremented.

There is also an effect that is not mentioned in the software flow graph. We were able to implement modulation as an effect. Modulation is a way to output two frequencies slightly out of tune. This provides a different sound for one note. For instance the frequency of middle C is 261 Hz. If we wanted to sound a modulated C sine wave, we would output a sine of 261 Hz *and* a sine of 259 Hz.

To implement modulation, we have 72 counters, two for each frequency. If a note is being played, we add the same saw tooth wave twice. The only difference is, one counter reaches its period one index before the other. This slightly changes the frequency of the second output, allowing for the modulation effect. If the modulation effect is selected, the multiplier is also affected. It is essentially cut in half because each note will have twice the amplitude (since two different waves are output for one note).

**3.8 Interfaces**

The primary interface between the various subsystems is I2C communication which takes place between the PIC24 and dsPIC33 microcontrollers. It is this interface which is responsible for the PIC24 sending the correct frequencies and effects to be played, as selected by the user, to the dsPIC33 so that the Audio Output subsystem can then generate the correct frequencies. For this communication, the PIC24 microcontroller was chosen to be the master and the dsPIC33 the slave. Thus, each of these microcontrollers has a distinct role to play in this I2C communication.

The PIC24 microcontroller is responsible for taking in all of the user inputs and generating a variable which contains which of the possible 36 frequencies and 4 effects are being implemented. This information is then stored by the PIC24 in a *long long* variable labeled “playThis” in which only 5 of the 8 bytes are used. Because the PIC24’s software for this project is implemented by polling and not interrupts, a new “playThis” variable is generated within each cycle of the main code of the PIC24, which is executed in full about every 1 ms. In this same cycle is the code to send the 5 crucial bytes of this variable to the dsPIC33 via I2C. These 5 bytes are extracted from the *long long*  variable through a series of bit shifts and bit masking. In the initialization of the PIC24, outside of the main program’s loop, the I2C module is enabled and the clock is set to run at 100 kHz. Then, within each loop of the main program, the PIC24 generates a start sequence, transmits the 7-bit address of the dsPIC followed by a “0” in order to write to the dsPIC, then transmits the 5 bytes of data corresponding to the used portion of the “playThis” variable, and finally generates the stop sequence. After each byte of data received, the PIC24 also waits for an “Acknowledge” bit from the dsPIC33 to verify that it is ready for the microcontroller to send the next byte. Thus, this entire sequence is repeated about every 1 ms.

Unlike the PIC24, the I2C module of the dsPIC33 is set to be interrupt driven. Thus, although the I2C module is enabled at the beginning of the code and its slave address is manually set, the dsPIC will not interact via I2C unless an interrupt is generated. This was done to ensure that the dsPIC could generate the audio output at a sufficient rate while also always remaining in sync the bytes being sent via I2C. In I2C a slave interrupt flag is raised every time an address has been transmitted over the I2C module or after the eighth bit of a data byte has been transmitted. Thus, each time either of these events occurred, the dsPIC enters the interrupt service routine. In this routine, the dsPIC would first check to see if the last byte sent was an address, in which case an acknowledge from the dsPIC was already automatically generated. Otherwise, if a data byte was sent, the dsPIC would read this byte from the receive buffer, thus sending an “Ack” to the master and opening the buffer for future receptions. Since it was known that the dsPIC would always receive 5 bytes in succession, a simple counter was incremented each time the interrupt service routine was called to determine which inputs the most recent byte corresponded to and to assign this received byte to the correct variable. After the counter reached 5, it would be reset to 1 on the next generated interrupt. Furthermore, this counter was reset each time the slave address byte was received in order to ensure that the master and slave were in sync in terms of the bytes before the 5 data bytes were sent. Thus, an I2C communication interface was successfully implemented between the PIC24 and the dsPIC to transfer the user selected inputs to the audio output subsystem.

**4 System Integration Testing**

The subsystems on the KEYtar were tested in small parts before being assembled into the final KEYtar. Subsystems were tested in connection to only one or two other subsystems, and only once all of the interconnections were tested was the entire system tested as a whole. Below is a list of the subsystem interconnections that were tested, as well as a brief explanation of how they were tested.

|  |  |
| --- | --- |
| Subsystem interconnection | Explanation |
| Sound related inputs to Audio Output | This subsystem interconnection was certainly the most significant. We had to connect the system that took as inputs all of the buttons (piano, chord, effect) and transmitted that data to the DSP chip, which ran the audio output algorithm to properly send the correct audio data to the DAC. To do this, we started small and worked our way up. Instead of starting with the full piano key set, we started by making the effects buttons produce sounds, and only a few sounds at a time. Then we expanded to making the entire piano produce notes. The next step was to include the chord buttons. Once the piano was working properly in the key of C, we included the key select assist mode and the chord select in standard mode. By doing this testing in pieces, we had a much easier time identifying where problems were when they inevitably occurred. |
| Key Select to Sound related inputs | The idea here was to test that out key select subsystem correctly modified the information from the sound related inputs (i.e. piano and chord buttons). The easiest way for us to do this was to put this step last, after we had already gotten the full piano and chord button board to produce the appropriate frequencies from the speaker. Once we had all of these other systems in place, testing this subsystem interconnection was as simple as changing the key while playing notes in assist mode and seeing in real-time as the note we were playing was tracking the key that we had as the assist setting. In assist mode, we expected the notes on the piano keys to change appropriately, and in standard mode we expected the notes on the piano keys to not change. We expected the same changes from the chord buttons regardless of the mode. |
| Key Select to User interface display | This was simple because of the lack of auto-assist mode. To test this interface, all that we needed to do was try the two modes (standard and assist) and spin the quadrature encoder knob and cycle through all the possible key options. |
| Entire KEYtar | To test the entire KEYtar, we simply played it a lot! We played all the buttons and tried all of the different keys. When al l the subsystems were put together and the KEYtar was working as expected, it was obvious that the subsystems were working together properly! |

By testing the entire KEYtar as a whole, we were able to evaluate how the overall system met the design requirements. Below is the list of design requirements and evaluation of whether or not the KEYtar met those requirements.

|  |  |
| --- | --- |
| Design Requirement | Did the KEYtar meet the requirement? |
| Must play music at audible level | Yes. The volume from the on-board speaker is as loud as a soft acoustic guitar. The external speaker connection also fulfills this requirement. |
| Must play the input from the user | Yes. |
| Must determine key from knob and microphone | No. The KEYtar can map a new key in assist-mode using the knob, but Auto-assist mode using the microphone has not been implemented. |
| Must be able to output sound to an external amplifier | Yes. Output is from a 3.5mm headphone jack. |
| Must not shock people | The KEYtar hasn’t shocked anyone as of 5/8/2013 |
| Must not splinter | The KEYtar hasn’t splintered as of 5/8/2013 |
| Must not cause fire | The KEYtar hasn’t caused a fire as of 5/8/2013 |
| Must not expend more than 50 watts | The KEYtar uses nowhere near 50 watts. The audio amplifier maxes out at 2 watts. Other power draw is small in comparison. |
| Must operate at least 30 minutes on battery power | No. There are no onboard batteries. |
| Must charge within one hour | N/A. There are no onboard batteries. |
| Must operate with wall plug | Yes. This works properly. |
| Must survive 300 charge cycles | N/A. No onboard batteries |
| Must not exceed $500 to design and produce | Yes. I do not have the exact numbers, but we spent far less than $500. |
| Must be less than 4’x2’x4” in dimension | Yes. Dimensions are <4’x2’x2” |
| Must be less than 12 pounds | Not sure. We did not have a scale to measure, but the KEYtar is not prohibitively heavy. |
| Must be intuitive to operate | Sort of. The KEYtar is not currently well-labeled since it is a prototype. I currently would not consider it intuitive to operate without some instructions. |
| Buttons must be easy to press | Sort of. Piano buttons are only slightly difficult to press because of the mediocre piano key construction. Chord buttons are very easy to press, as are the effects buttons. |
| LEDs must be bright enough to see | Yes. |
| Must be sturdy and withstand normal usage | Sort of. The KEYtar is very sturdy for a prototype, but some improvements are needed to its strap holders. It is not sturdy enough to be sold yet. |

**5 User Manual / Installation Manual**

The KEYtar is fairly easy to use, but usage questions may still exist. Installation and use questions can hopefully be answered by the guide below.

Installation and Setup

Congratulations on your new shark-shaped KEYtar! Your new KEYtar comes fully assembled and setup is easy. To turn the KEYtar on, simply plug a DC power source of 7.5 volts into the Power Jack (see photo above). Turn the volume knob to the desired volume, and you’re ready to go! To play sound out of the speaker, make sure there is nothing plugged into the 3.5mm jack. To play sounds from an external sound system of headphones, simply plug them into the 3.5mm jack, and the sound output will automatically switch from the speaker to the 3.5mm jack.

How to tell if the KEYtar is working

First, install and setup the KEYtar as described above. If the Key Select Screen LEDs are lit, the KEYtar is on. Play the piano keys and chords buttons. If the KEYtar produces the expected sounds, the KEYtar is working properly! If the KEYtar is not working properly, see the troubleshooting guide.

Usage

If your KEYtar is working properly, use the following information to get the most out of your KEYtar experience.

* The green buttons are the effects buttons. In order from left to right (when viewed from the musician) no effect, modulation 1, modulation 2, no effect, no effect, no effect.
* In Standard mode (chosen by the mode switch), the piano keys play notes just as a normal piano would. The chord buttons play the I, II, III, IV, V VI, VII chords in whatever key is being displayed on the Key Select Screen.
* In Assist mode (chosen by the mode switch), the piano keys are mapped to the key that is shown on the key select screen. The note shown on the screen is the root note, and playing the C key on the piano with produce the root note. The black keys are disabled in Assist mode. The chord buttons in Assist mode function the same way that they do in Standard mode.

Troubleshooting Guide

* Problem
  + Solutions
* KEYtar won’t turn on
  + Is the power board working? Does the voltage going to the microcontroller board from the power board equal GND, 3.3V, and 5V? If not, the power board may be broken, or the DC power supply might not be working. If the voltage is correct, there is a chance that the microcontroller is not working. If the voltages going to the microcontroller are correct, return the KEYtar to the manufacturer for more in-depth troubleshooting.
* KEYtar is on, but won’t output audio
  + Is the volume turned up? Turn the knob counter-clockwise to increase the volume.
  + Try using the headphone jack as well as the speaker. Is there sound coming from either output?
  + Check the wire connections between the DSP board and the Microcontroller board. Are all the connections in place?
  + Still not working? The audio amp may not be working. Send the KEYtar to the manufacturer for advanced troubleshooting.
* Key select knob is not working
  + This happens frequently. The simple solution is to turn the knob slower. It will work if you turn it slow enough. Trust me.
* KEYtar keeps resetting randomly to the key of C
  + Yea, about that…………. It does that.

**6 To – Market Design Changes**

Before the KEYtar is ready for the market, quite a few improvements would need to be made. Below is a brief list, followed by a description of each improvement.

1. Improved sound effects
2. Rechargeable battery
3. Auto-assist mode
4. Improved Shark body

*Improved Sound Effects*

Currently, the KEYtar has essentially two sound effects. The first is its standard sound, which is produced using a ramp wave. The second sound effect is a type of modulation created by simultaneously playing two slightly-out-of-tune ramp waves.

Before bringing the KEYtar to market, we would add more sound effects. Different instruments could be created using our wavetable concept (where the wavetables are stored in the memory that we could not get working). Other effects would need to be created by running the output through some sort of digital filter. Using these two methods, we think that we could create many effects, including reverb, echo, delay, phasers and modulation, distortion, and all types of instrument sounds. Currently, the KEYtar only has 6 effects buttons (and space for 8), so any major improvement to effects would most likely require some sort of LCD screen interface so that the user can intelligently figure out what effects they are using.

*Rechargeable battery*

The KEYtar currently runs off of a DC input produced by a DC (wall jack) power supply. This is inconvenient because the user is closely tied to their nearest power outlet, and the KEYtar unfortunately becomes much less portable. The Power Board mounted on the KEYtar has inputs or voltage produced from a battery, and making the KEYtar battery powered would be as simple as mounting the batteries somewhere on the KEYtar. However, frequently changing KEYtar batteries would be a pain as well. The ideal solution would be to have rechargeable batteries.

Rechargeable batteries could be mounted on the back of the KEYtar with a charging connection in the same place that the current power board is mounted. We would most likely use Lithium Ion batteries because of their high power density and long life.

*Auto-assist mode*

In its current state, the KEYtar only has standard and assist modes. These are two of the three crucial modes of the KEYtar, but the original idea of the KEYtar (an instrument that anyone can pick up and play with friends quickly) gets lost without auto-assist mode.

Auto-assist mode allows the user to listen to ambient sounds to determine the key that his or her friends are playing in. It then automatically sets the KEYtar to play in that key. All of the hardware is currently in place to implement auto-assist mode, but the software has not been successfully written. The KEYtar that gets sent to market would have this software written and Auto-assist mode would be functional.

*Improved Shark body*

The shark-shaped body of the KEYtar is currently very cool, but it could use quite a few functional improvements to make it fit for consumer use.

First of all, the circuit boards need covers so that they are not exposed. Also, the power board on the side of the shark is not adequately covered, and does not have adequate cooling. There is also not place on the shark for mounting batteries.

The other major part of the shark that could use functional improvement is the set of piano keys. The shark that gets sent to market would need an entirely redesigned piano key area. The keys wouldn’t be made out of a 2x4. I think the keys would be less like real piano keys, and more like sweet KEYtar buttons. For a functional prototype, the piano keys were playable, but there is where the most significant body improvements need to be made.

**7 Conclusions**

The KEYtar meets most of the requirements that were initially proposed. The requirements that were not met did not prevent a functional system from being constructed. In the end, the subsystems were functional and could be combined into an overall working system. The KEYtar was able to play the correct notes based on the user’s selection both on the piano keys and on the chord buttons. The three options for sound effects were also functional based on the user selection. Additionally, the audio output was operating both for the on-board speaker and the headphone jack.

Furthermore, Standard Mode and Assist Mode were both functional. The key select knob allowed the user to choose a key in which to play. Additionally, the sixteen segment LED correctly displayed the key in which the user was playing. Thus, the KEYtar was able to partially fulfill its purpose to aid those who are not as musically competent to play music in assist mode. Finally, the body of the KEYtar met the requirements and was appealing in color and shape.

Some of the unfulfilled requirements could be easily met as next steps. The batteries could be attached to the outside of the KEYtar body and plugged into the power board. The hardware for the auto-assist mode is built and a prototype algorithm is already prepared, so implementing the auto-assist mode would be a practical next step that would significantly help the KEYtar in fulfilling its purpose to help those without musical talent. Another step would be to implement more sound effects and to achieve a functional memory device in which to store wavetables for multiple audio selections.

Overall, the KEYtar was a functioning system that provided great music and enjoyment to those who played it.

**8 Appendices**

**Board Schematics and Layouts:**

(see http://seniordesign.ee.nd.edu/2013/Design%20Teams/keytar/index.html)

|  |
| --- |
|  |
| **The PIC24 microcontroller board schematic.** |

|  |
| --- |
|  |
| **The PIC24 microcontroller board layout.** |

|  |
| --- |
|  |
| **The dsPIC33F microcontroller schematic.** |

|  |
| --- |
|  |
| **The dsPIC33F microcontroller layout.** |

|  |
| --- |
|  |
| **The power board schematic.** |

|  |
| --- |
|  |
| **The power board layout.** |

|  |
| --- |
|  |
| **The piano button board schematic.** |

|  |
| --- |
|  |
| **The piano button board layout.** |

|  |
| --- |
|  |
| **The chord button board schematic.** |

|  |
| --- |
|  |
| **The chord button board layout.** |

**Software used to program dsPIC board:**

/\*

\* File: DACmain\_withInterrupts\_v3

\* Author: Zach Stauder

\*

\* Created on Mary 2, 2013, 7:10 PM

\*/

#include <stdio.h>

#include <stdlib.h>

#include<xc.h>

#include<math.h>

#include<i2c.h>

#include "freqValues.h"

/\*

\*

\*/

//

\_FOSC(FCKSM\_CSECMD & OSCIOFNC\_OFF & POSCMD\_NONE);

\_FOSCSEL(FNOSC\_FRC);

//#define PORTCbits.RC8 = 1 SHUTDOWN\_OFF

//#define PORTCbits.RC8 = 1 SHUTDOWN\_ON

//long long analyzeInputs(long long inputs) {

//

// if((inputs&0x03)==0 { // standard mode (bits 0-1 for mode)

//

// long long outFreqs = (0x0000000007FFFFFC | inputs); // bits 2-26 for piano keys

// outFreqs = outFreqs|

// }

//}

//void initRamps(){

//

//}

unsigned char getI2CByte(){

unsigned char data;

while(!I2C1STATbits.RBF);

if(I2C1STATbits.D\_A==0) // Wait until last byte was address

data = I2C1RCV;

return data;

}

void initiateI2C(){

I2C1BRG = 156; //132 // I21BRG = (1/I2C\_CLK - 130ns)\*SYS\_CLK-2

// made clock 400 kHz // 156 makes it 100 kHz // made clock 400 kHz

I2C1CONbits.I2CSIDL = 0; // operate in idle mode

I2C1CONbits.IPMIEN = 0; // ?????

I2C1CONbits.A10M = 0; // 7 bit slave address

I2C1ADD = 0b1010101; // set 7 bit slave address

I2C1CONbits.I2CEN = 1; // enable I2C module

}

void initiateInterrupts(void){

INTCON1bits.NSTDIS = 1; //Nested interrupts are not desired

IEC1bits.SI2C1IE = 1; //Enable Slave Interrupt for I2C

IFS1bits.SI2C1IF = 0; //Clear Slave Interrupt Flag

}

unsigned char address\_rec = 0;

unsigned char buttonStates1 = 0;

unsigned char buttonStates2 = 0;

unsigned char byteCount = 0;

unsigned int freqs[36];

unsigned int eff[4];

unsigned char selection = 0;

unsigned char recByte1 = 0;

unsigned char recByte2 = 0;

unsigned char recByte3 = 0;

unsigned char recByte4 = 0;

unsigned char recByte5 = 0;

unsigned int numFreqs = 0;

unsigned int multFactor = 1;

unsigned int modulation = 0;

void \_\_attribute\_\_((\_\_interrupt\_\_, no\_auto\_psv)) \_SI2C1Interrupt(void)

{

IFS1bits.SI2C1IF = 0; // Clear Interrupt Flag

if(I2C1STATbits.D\_A==0){ //If last byte received was an address

address\_rec = I2C1RCV; //Unused variable to clear receive buffer

byteCount = 0;

}

else if(I2C1STATbits.RBF==1){ //if receive buffer is full

if(byteCount==0){

recByte1 = I2C1RCV; //Set data byte

byteCount = byteCount+1;

}

else if(byteCount==1){

recByte2 = I2C1RCV; //Set data byte

byteCount = byteCount+1;

}

else if(byteCount==2){

recByte3 = I2C1RCV; //Set data byte

byteCount = byteCount+1;

}

else if(byteCount==3){

recByte4 = I2C1RCV; //Set data byte

byteCount = byteCount+1;

}

else{

recByte5 = I2C1RCV;

byteCount = 0;

}

freqs[0] = (recByte1&128)==128;

freqs[1] = (recByte1&64)==64;

freqs[2] = (recByte1&32)==32;

freqs[3] = (recByte1&16)==16;

freqs[4] = (recByte1&8)==8;

freqs[5] = (recByte1&4)==4;

freqs[6] = (recByte1&2)==2;

freqs[7] = recByte1&1;

freqs[8] = (recByte2&128)==128;

freqs[9] = (recByte2&64)==64;

freqs[10] = (recByte2&32)==32;

freqs[11] = (recByte2&16)==16;

freqs[12] = (recByte2&8)==8;

freqs[13] = (recByte2&4)==4;

freqs[14] = (recByte2&2)==2;

freqs[15] = recByte2&1;

freqs[16] = (recByte3&128)==128;

freqs[17] = (recByte3&64)==64;

freqs[18] = (recByte3&32)==32;

freqs[19] = (recByte3&16)==16;

freqs[20] = (recByte3&8)==8;

freqs[21] = (recByte3&4)==4;

freqs[22] = (recByte3&2)==2;

freqs[23] = recByte3&1;

freqs[24] = (recByte4&128)==128;

freqs[25] = (recByte4&64)==64;

freqs[26] = (recByte4&32)==32;

freqs[27] = (recByte4&16)==16;

freqs[28] = (recByte4&8)==8;

freqs[29] = (recByte4&4)==4;

freqs[30] = (recByte4&2)==2;

freqs[31] = recByte4&1;

freqs[32] = (recByte5&128)==128;

freqs[33] = (recByte5&64)==64;

freqs[34] = (recByte5&32)==32;

freqs[35] = (recByte5&16)==16;

eff[0] = (recByte5&8)==8;

eff[1] = (recByte5&4)==4;

eff[2] = (recByte5&2)==2;

eff[3] = recByte5&1;

numFreqs = 0;

int i = 0;

for(i=0;i<36;i++){

numFreqs = numFreqs+freqs[i];

}

if(eff[0]==1){

modulation=0;

}

else if(eff[1]==1){

modulation=1;

}

else if(eff[2]==1){

modulation=3;

}

else{

modulation=0;

}

switch (modulation){

case 0:

if(numFreqs==1){

multFactor=8;

}

else if(numFreqs==2){

multFactor=5;

}

else if(numFreqs==3){

multFactor=4;

}

else if(numFreqs==4){

multFactor=3;

}

else if(numFreqs==5){

multFactor=3;

}

else if(numFreqs==6){

multFactor=2;

}

else if(numFreqs==7){

multFactor=2;

}

else{

multFactor=1;

}

break;

default:

if(numFreqs==1){

multFactor=4;

}

else if(numFreqs==2){

multFactor=3;

}

else if(numFreqs==3){

multFactor=2;

}

else if(numFreqs==4){

multFactor=2;

}

else{

multFactor=1;

}

break;

}

}

}

int main(int argc, char\*\* argv) {

//set auxillary clock

OSCCONbits.COSC = 0b001; // set clock to fast RC oscillator 7.37 MHz, divide by n, PLL

OSCCONbits.NOSC = 0b001; // new oscillator selection bits FRC oscillator, divide n, PLL (pg 147)

PLLFBDbits.PLLDIV = 59; // clock multiplier n+2

CLKDIVbits.FRCDIV = 0; // clock divider = 1

CLKDIVbits.PLLPOST = 0; // divide clock by two (2^(n+1))

CLKDIVbits.PLLPRE = 2; // divide clock by four (n+2)

//Fosc = 7.37MHz\*(59+2)/(4\*2\*1) = 56.196MHz

// need 256\*44.1kHz\*5 = 56448000

OSCTUNbits.TUN = 1; // tune clock up .375 percent

//Fosc = 7.37\*1.00375\*(59+2)/(4\*2\*1) = 56.4 MHz

// Clock switch to incorporate PLL

\_\_builtin\_write\_OSCCONH(0x01); // Initiate Clock Switch to

// FRC with PLL (NOSC=0b001)

\_\_builtin\_write\_OSCCONL(0x01); // Start clock switching

while (OSCCONbits.COSC != 0b001); // Wait for Clock switch to occur

ACLKCONbits.AOSCMD = 0; // auxillary clock disabled

ACLKCONbits.SELACLK = 0; // auxillary clock selects FOSC as source

ACLKCONbits.APSTSCLR = 7; // divide by 1

DAC1CONbits.DACSIDL = 1; // stop DAC module in idle mode

DAC1CONbits.AMPON = 0; // disable analog ouput amplifier in sleep/idle mode

DAC1CONbits.FORM = 1; // signed ints, midpoint at 0x8000

DAC1CONbits.DACEN = 1; // enable DAC

DAC1CONbits.DACFDIV = 9; // divide auxillary clock by 10 for DAC clock

// The clock needs to be 256 times the sampling rate

// making the clock 11.289 MHz

// 56.4 Mhz/10/256\*2 = 44100 Hz

DAC1STATbits.LOEN = 0; // left channel output DAC disabled

DAC1STATbits.LMVOEN = 0; // left channel midpoint DAC output disable

DAC1STATbits.LITYPE = 0; // left channel interrupt if FIFO empty

//DAC1STATbits.LFULL = 1 if full, 0 if not full, read only

//DAC1STATbits.LEMPTY = 1 if empty, 0 if not, read only

DAC1STATbits.ROEN = 1; // right channel output DAC enabled

DAC1STATbits.RMVOEN = 1; // right channel midpoint DAC output enable

DAC1STATbits.RITYPE = 1; // right channel interrupt if FIFO empty

TRISCbits.TRISC8 = 0; // OUTPUT

PORTCbits.RC8 = 1; // SET AS HIGH, SHUTDOWN OFF

//DAC1STATbits.RFULL = 1 if full, 0 if not full, read only

//DAC1STATbits.REMPTY = 1 if empty, 0 if not, read only

DAC1DFLT = 0x0000; // default DAC output

DAC1RDAT = 0x0000; // right data register

DAC1LDAT = 0x0000; // left data register

//

int ramp[C1pts];

//int increments[36] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

//int mods[36] = {C1pts,C1Spts,D1pts,D1Spts,E1pts,F1pts,F1Spts,G1pts,G1Spts,A1pts,A1Spts,B1pts,

// C2pts,C2Spts,D2pts,D2Spts,E2pts,F2pts,F2Spts,G2pts,G2Spts,A2pts,A2Spts,B2pts,

// C3pts,C3Spts,D3pts,D3Spts,E3pts,F3pts,F3Spts,G3pts,G3Spts,A3pts,A3Spts,B3pts};

//unsigned int freqs[36];

int c1i=0;

int c1si=0;

int d1i=0;

int d1si=0;

int e1i=0;

int f1i=0;

int f1si=0;

int g1i=0;

int g1si=0;

int a1i=0;

int a1si=0;

int b1i=0;

int c2i=0;

int c2si=0;

int d2i=0;

int d2si=0;

int e2i=0;

int f2i=0;

int f2si=0;

int g2i=0;

int g2si=0;

int a2i=0;

int a2si=0;

int b2i=0;

int c3i=0;

int c3si=0;

int d3i=0;

int d3si=0;

int e3i=0;

int f3i=0;

int f3si=0;

int g3i=0;

int g3si=0;

int a3i=0;

int a3si=0;

int b3i=0;

int c1o=0;

int c1so=0;

int d1o=0;

int d1so=0;

int e1o=0;

int f1o=0;

int f1so=0;

int g1o=0;

int g1so=0;

int a1o=0;

int a1so=0;

int b1o=0;

int c2o=0;

int c2so=0;

int d2o=0;

int d2so=0;

int e2o=0;

int f2o=0;

int f2so=0;

int g2o=0;

int g2so=0;

int a2o=0;

int a2so=0;

int b2o=0;

int c3o=0;

int c3so=0;

int d3o=0;

int d3so=0;

int e3o=0;

int f3o=0;

int f3so=0;

int g3o=0;

int g3so=0;

int a3o=0;

int a3so=0;

int b3o=0;

int factor=1;

int index;

for (index = 0; index<C1pts; index++) {

ramp[index]=factor\*index;

}

// for (index = 0; index<D1pts; index++) {

// ramp[index]=factor\*index;

// }

// for (index = 0; index<C1pts; index++) {

// ramp[index]=factor\*index;

// }

// for (index = 0; index<C1pts; index++) {

// ramp[index]=factor\*index;

// }

// for (index = 0; index<C1pts; index++) {

// ramp[index]=factor\*index;

// }

// for (index = 0; index<C1pts; index++) {

// ramp[index]=factor\*index;

// }

// for (index = 0; index<C1pts; index++) {

// ramp[index]=factor\*index;

// }

// for (index = 0; index<C1pts; index++) {

// ramp[index]=factor\*index;

// }

// for (index = 0; index<C1pts; index++) {

// ramp[index]=factor\*index;

// }

initiateI2C();

initiateInterrupts();

TRISBbits.TRISB6 = 0;

LATBbits.LATB6 = 0;

long yout;

long long selection = 0;

int time=0;

while(1) {

// selection1 = buttonStates1;

selection = buttonStates2;

/\* if (time==50) {

//selection = getI2CByte();

freqs[0] = selection&1;

freqs[1] = (selection&2)==2;

freqs[2] = (selection&4)==4;

freqs[3] = (selection&8)==8;

freqs[4] = (selection&16)==16;

freqs[5] = (selection&32)==32;

freqs[6] = (selection&64)==64;

freqs[7] = (selection&128)==128;

time=0;

}

\*/

//LATCbits.LATC6=1;

// yout = C1factor\*freqs[0]\*ramp[c1i] + D1factor\*freqs[2]\*ramp[d1i] +

// E1factor\*freqs[4]\*ramp[e1i] + F1factor\*freqs[5]\*ramp[f1i]+

// G1factor\*freqs[7]\*ramp[g1i] + A1factor\*0\*ramp[a1i] +

// B1factor\*0\*ramp[b1i] + C2factor\*0\*ramp[c2i] +

// C1Sfactor\*freqs[1]\*ramp[c1si] + D1Sfactor\*freqs[3]\*ramp[d1si] +

// F1Sfactor\*freqs[6]\*ramp[f1si] + G1Sfactor\*0\*ramp[g1si] +

// A1Sfactor\*0\*ramp[a1si];

// yout = C1factor\*freqs[0]\*ramp[c1i++] + C1Sfactor\*freqs[1]\*ramp[c1si++] +

// D1factor\*freqs[2]\*ramp[d1i++] + D1Sfactor\*freqs[3]\*ramp[d1si++] +

// E1factor\*freqs[4]\*ramp[e1i++] + F1factor\*freqs[5]\*ramp[f1i++]+

// F1Sfactor\*freqs[6]\*ramp[f1si++]+G1factor\*freqs[7]\*ramp[g1i++] +

// G1Sfactor\*0\*ramp[g1si++]+A1factor\*0\*ramp[a1i++] +

// A1Sfactor\*0\*ramp[a1i] + B1factor\*0\*ramp[b1i++] + C2factor\*0\*ramp[c1i++];

// C2factor\*freqs[12]\*ramp[c2i++];

//yout=5000\*(yref[(index%floor(SAMPFREQ/C1))\*pointref/floor(SAMPFREQ/C1)]+yref[(index%\*pointref/floor(SAMPFREQ/E1)]);

while(DAC1STATbits.RFULL == 1);

DAC1RDAT = yout;

/\*if (time==200) {

//selection = getI2CByte();

freqs[0] = selection&1;

freqs[1] = (selection&2)==2;

freqs[2] = (selection&4)==4;

freqs[3] = (selection&8)==8;

freqs[4] = (selection&16)==16;

freqs[5] = (selection&32)==32;

freqs[6] = (selection&64)==64;

freqs[7] = (selection&128)==128;

time=0;

}\*/

yout=0;

if (freqs[0]==1){

c1i++;

if (c1i==C1pts)

c1i=0;

yout=yout+C1factor\*ramp[c1i]\*multFactor;

if (!modulation==0){

c1o++;

if (c1o==C1pts-modulation)

c1o=0;

yout=yout+C1factor\*ramp[c1o]\*multFactor;

}

}

if (freqs[1]==1) {

c1si++;

if (c1si==C1Spts)

c1si=0;

yout=yout+C1Sfactor\*ramp[c1si]\*multFactor;

if (!modulation==0){

c1so++;

if (c1so==C1Spts-modulation)

c1so=0;

yout=yout+C1Sfactor\*ramp[c1so]\*multFactor;

}

}

if (freqs[2]==1) {

d1i++;

if (d1i==D1pts)

d1i=0;

yout=yout+D1Sfactor\*ramp[d1i]\*multFactor;

if (!modulation==0){

d1o++;

if (d1o==D1pts-modulation)

d1o=0;

yout=yout+D1factor\*ramp[d1o]\*multFactor;

}

}

if (freqs[3]==1) {

d1si++;

if (d1si==D1Spts)

d1si=0;

yout=yout+D1Sfactor\*ramp[d1si]\*multFactor;

if (!modulation==0){

d1so++;

if (d1so==D1Spts-modulation)

d1so=0;

yout=yout+D1Sfactor\*ramp[d1so]\*multFactor;

}

}

if (freqs[4]==1) {

e1i++;

if (e1i==E1pts)

e1i=0;

yout=yout+E1factor\*ramp[e1i]\*multFactor;

if (!modulation==0){

e1o++;

if (e1o==E1pts-modulation)

e1o=0;

yout=yout+E1factor\*ramp[e1o]\*multFactor;

}

}

if (freqs[5]==1) {

f1i++;

if (f1i==F1pts)

f1i=0;

yout=yout+F1factor\*ramp[f1i]\*multFactor;

if (!modulation==0){

f1o++;

if (f1o==F1pts-modulation)

f1o=0;

yout=yout+F1factor\*ramp[f1o]\*multFactor;

}

}

if (freqs[6]==1) {

f1si++;

if (f1si==F1Spts)

f1si=0;

yout=yout+F1Sfactor\*ramp[f1si]\*multFactor;

if (!modulation==0){

f1so++;

if (f1so==F1Spts-modulation)

f1so=0;

yout=yout+F1Sfactor\*ramp[f1so]\*multFactor;

}

}

if (freqs[7]==1) {

g1i++;

if (g1i==G1pts)

g1i=0;

yout=yout+G1factor\*ramp[g1i]\*multFactor;

if (!modulation==0){

g1o++;

if (g1o==G1pts-modulation)

g1o=0;

yout=yout+G1factor\*ramp[g1o]\*multFactor;

}

}

if (freqs[8]==1) {

g1si++;

if (g1si==G1Spts)

g1si=0;

yout=yout+G1Sfactor\*ramp[g1si]\*multFactor;

if (!modulation==0){

g1so++;

if (g1so==G1Spts-modulation)

g1so=0;

yout=yout+G1Sfactor\*ramp[g1so]\*multFactor;

}

}

if (freqs[9]==1) {

a1i++;

if (a1i==A1pts)

a1i=0;

yout=yout+A1factor\*ramp[a1i]\*multFactor;

if (!modulation==0){

a1o++;

if (a1o==A1pts-modulation)

a1o=0;

yout=yout+A1factor\*ramp[a1o]\*multFactor;

}

}

if (freqs[10]==1) {

a1si++;

if (a1si==A1Spts)

a1si=0;

yout=yout+A1Sfactor\*ramp[a1si]\*multFactor;

if (!modulation==0){

a1so++;

if (a1so==A1Spts-modulation)

a1so=0;

yout=yout+A1Sfactor\*ramp[a1so]\*multFactor;

}

}

if (freqs[11]==1) {

b1i++;

if (b1i==B1pts)

b1i=0;

yout=yout+B1factor\*ramp[b1i]\*multFactor;

if (!modulation==0){

b1o++;

if (b1o==B1pts-modulation)

b1o=0;

yout=yout+B1factor\*ramp[b1o]\*multFactor;

}

}

if (freqs[12]==1) {

c2i++;

if (c2i==C2pts)

c2i=0;

yout=yout+C2factor\*ramp[c2i]\*multFactor;

if (!modulation==0){

c2o++;

if (c2o==C2pts-modulation)

c2o=0;

yout=yout+C2factor\*ramp[c2o]\*multFactor;

}

}

if (freqs[13]==1) {

c2si++;

if (c2si==C2Spts)

c2si=0;

yout=yout+C2Sfactor\*ramp[c2si]\*multFactor;

if (!modulation==0){

c2so++;

if (c2so==C2Spts-modulation)

c2so=0;

yout=yout+C2Sfactor\*ramp[c2so]\*multFactor;

}

}

if (freqs[14]==1) {

d2i++;

if (d2i==D2pts)

d2i=0;

yout=yout+D2factor\*ramp[d2i]\*multFactor;

if (!modulation==0){

d2o++;

if (d2o==D2pts-modulation)

d2o=0;

yout=yout+D2factor\*ramp[d2o]\*multFactor;

}

}

if (freqs[15]==1) {

d2si++;

if (d2si==D2Spts)

d2si=0;

yout=yout+D2Sfactor\*ramp[d2si]\*multFactor;

if (!modulation==0){

d2so++;

if (d2so==D2Spts-modulation)

d2so=0;

yout=yout+D2Sfactor\*ramp[d2so]\*multFactor;

}

}

if (freqs[16]==1) {

e2i++;

if (e2i==E2pts)

e2i=0;

yout=yout+E2factor\*ramp[e2i]\*multFactor;

if (!modulation==0){

e2o++;

if (e2o==E2pts-modulation)

e2o=0;

yout=yout+E2factor\*ramp[e2o]\*multFactor;

}

}

if (freqs[17]==1) {

f2i++;

if (f2i==F2pts)

f2i=0;

yout=yout+F2factor\*ramp[f2i]\*multFactor;

if (!modulation==0){

f2o++;

if (f2o==F2pts-modulation)

f2o=0;

yout=yout+F2factor\*ramp[f2o]\*multFactor;

}

}

if (freqs[18]==1) {

f2si++;

if (f2si==F2Spts)

f2si=0;

yout=yout+F2Sfactor\*ramp[f2si]\*multFactor;

if (!modulation==0){

f2so++;

if (f2so==F2Spts-modulation)

f2so=0;

yout=yout+F2Sfactor\*ramp[f2so]\*multFactor;

}

}

if (freqs[19]==1) {

g2i++;

if (g2i==G2pts)

g2i=0;

yout=yout+G2factor\*ramp[g2i]\*multFactor;

if (!modulation==0){

g2o++;

if (g2o==G2pts-modulation)

g2o=0;

yout=yout+G2factor\*ramp[g2o]\*multFactor;

}

}

if (freqs[20]==1) {

g2si++;

if (g2si==G2Spts)

g2si=0;

yout=yout+G2Sfactor\*ramp[g2si]\*multFactor;

if (!modulation==0){

g2so++;

if (g2so==G2Spts-modulation)

g2so=0;

yout=yout+G2Sfactor\*ramp[g2so]\*multFactor;

}

}

if (freqs[21]==1) {

a2i++;

if (a2i==A2pts)

a2i=0;

yout=yout+A2factor\*ramp[a2i]\*multFactor;

if (!modulation==0){

a2o++;

if (a2o==A2pts-modulation)

a2o=0;

yout=yout+A2factor\*ramp[a2o]\*multFactor;

}

}

if (freqs[22]==1) {

a2si++;

if (a2si==A2Spts)

a2si=0;

yout=yout+A2Sfactor\*ramp[a2si]\*multFactor;

if (!modulation==0){

a2so++;

if (a2so==A2Spts-modulation)

a2so=0;

yout=yout+A2Sfactor\*ramp[a2so]\*multFactor;

}

}

if (freqs[23]==1) {

b2i++;

if (b2i==B2pts)

b2i=0;

yout=yout+B2factor\*ramp[b2i]\*multFactor;

if (!modulation==0){

b2o++;

if (b2o==B2pts-modulation)

b2o=0;

yout=yout+B2factor\*ramp[b2o]\*multFactor;

}

}

if (freqs[24]==1) {

c3i++;

if (c3i==C3pts)

c3i=0;

yout=yout+C3factor\*ramp[c3i]\*multFactor;

if (!modulation==0){

c3o++;

if (c3o==C3pts-modulation)

c3o=0;

yout=yout+C3factor\*ramp[c3o]\*multFactor;

}

}

if (freqs[25]==1) {

c3si++;

if (c3si==C3Spts)

c3si=0;

yout=yout+C3Sfactor\*ramp[c3si]\*multFactor;

if (!modulation==0){

c3so++;

if (c3so==C3Spts-modulation)

c3so=0;

yout=yout+C3Sfactor\*ramp[c3so]\*multFactor;

}

}

if (freqs[26]==1) {

d3i++;

if (d3i==D3pts)

d3i=0;

yout=yout+D3factor\*ramp[d3i]\*multFactor;

if (!modulation==0){

d3o++;

if (d3o==D3pts-modulation)

d3o=0;

yout=yout+D3factor\*ramp[d3o]\*multFactor;

}

}

if (freqs[27]==1) {

d3si++;

if (d3si==D3Spts)

d3si=0;

yout=yout+D3Sfactor\*ramp[d3si]\*multFactor;

if (!modulation==0){

d3so++;

if (d3so==D3Spts-modulation)

d3so=0;

yout=yout+D3Sfactor\*ramp[d3so]\*multFactor;

}

}

if (freqs[28]==1) {

e3i++;

if (e3i==E3pts)

e3i=0;

yout=yout+E3factor\*ramp[e3i]\*multFactor;

if (!modulation==0){

e3o++;

if (e3o==E3pts-modulation)

e3o=0;

yout=yout+E3factor\*ramp[e3o]\*multFactor;

}

}

if (freqs[29]==1) {

f3i++;

if (f3i==F3pts)

f3i=0;

yout=yout+F3factor\*ramp[f3i]\*multFactor;

if (!modulation==0){

f3o++;

if (f3o==F3pts-modulation)

f3o=0;

yout=yout+F3factor\*ramp[f3o]\*multFactor;

}

}

if (freqs[30]==1) {

f3si++;

if (f3si==F3Spts)

f3si=0;

yout=yout+F3Sfactor\*ramp[f3si]\*multFactor;

if (!modulation==0){

f3so++;

if (f3so==F3Spts-modulation)

f3so=0;

yout=yout+F3Sfactor\*ramp[f3so]\*multFactor;

}

}

if (freqs[31]==1) {

g3i++;

if (g3i==G3pts)

g3i=0;

yout=yout+G3factor\*ramp[g3i]\*multFactor;

if (!modulation==0){

g3o++;

if (g3o==G3pts-modulation)

g3o=0;

yout=yout+G3factor\*ramp[g3o]\*multFactor;

}

}

if (freqs[32]==1) {

g3si++;

if (g3si==G3Spts)

g3si=0;

yout=yout+G3Sfactor\*ramp[g3si]\*multFactor;

if (!modulation==0){

g3so++;

if (g3so==G3Spts-modulation)

g3so=0;

yout=yout+G3Sfactor\*ramp[g3so]\*multFactor;

}

}

if (freqs[33]==1) {

a3i++;

if (a3i==A3pts)

a3i=0;

yout=yout+A3factor\*ramp[a3i]\*multFactor;

if (!modulation==0){

a3o++;

if (a3o==A3pts-modulation)

a3o=0;

yout=yout+A3factor\*ramp[a3o]\*multFactor;

}

}

if (freqs[34]==1) {

a3si++;

if (a3si==A3Spts)

a3si=0;

yout=yout+A3Sfactor\*ramp[a3si]\*multFactor;

if (!modulation==0){

a3so++;

if (a3so==A3Spts-modulation)

a3so=0;

yout=yout+A3Sfactor\*ramp[a3so]\*multFactor;

}

}

if (freqs[35]==1) {

b3i++;

if (b3i==B3pts)

b3i=0;

yout=yout+B3factor\*ramp[b3i]\*multFactor;

if (!modulation==0){

b3o++;

if (b3o==B3pts-modulation)

b3o=0;

yout=yout+B3factor\*ramp[b3o]\*multFactor;

}

}

time=time+1;

}

return (EXIT\_SUCCESS);

}

**Software used to program PIC24 board:**

/\*

\* File: PIC24\_ButtonsWithI2C

\* Author: Zach

\*

\* Created on May 1, 2013, 02:29 AM

\*/

#include <stdio.h>

#include <stdlib.h>

#include <PIC24F\_plib.h>

#include <stdint.h>

#include <p24FJ128GA010.h>

#include <i2c.h>

#include "MiscLabels.h"

#include "EffectsButtons.h"

#include "SixteenSegmentLED.h"

#include "ChordButtons.h"

#include "PianoButtons.h"

#include "TRIS\_Initialization.c"

/\*

\*

\*/

#define numPins 45 // number of pins that are used as inputs

uint8\_t output[numPins] = {0}; // initialize all output values to 0

static uint8\_t oldvalue[numPins] = {0}; // initialize all oldvalues to 0

static uint8\_t flag[numPins] = {0}; // initialize all flag variables to 0

static uint8\_t flag2[8] = {0}; // initialize all flag2 variables to 0

static uint8\_t LEDon[8] = {0}; // initialize all LEDon variables to 0

static uint8\_t oldA = 0;

uint8\_t key = 1;

uint8\_t temp; // initialize a temporary variable for calculations

#define A 8

#define B 9

#define buttonPress 1 // pressing the button grounds the pin

#define t2intflag IFS0bits.T2IF // interupt flag bit for timer 2

#define i2cflag IFS1bits. //I2C

#define receive\_reg I2C1RCV //I2C

long long playThis = 0x0000000000000000;

\_CONFIG1( JTAGEN\_OFF); //Turn off JTAG so A0 and A1 are Digital I/O pins

unsigned char sendByte1 = 0;

unsigned char sendByte2 = 0;

unsigned char sendByte3 = 0;

unsigned char sendByte4 = 0;

unsigned char sendByte5 = 0;

void initiateClock(void){

OSCCONbits.COSC = 0b001; // sets system clock to Fosc and PLL

CLKDIVbits.RCDIV = 0b000; // divides clk by 1 = 8 MHz

/\* Clock appears to operate at 16 MHz\*/

}

void initiateTimer() {

T2CON = 0x0; // clear control register & stop timer

// T2CON<15> enables timer (0=off, 1=on)

// T2CON<1> = 0 selects PBCLK source (TCS)

// T2CON<7> = 0 selectrs PBCLK source (TGATE)

// set prescale value to 256

T2CONbits.TCKPS1 = 1;

T2CONbits.TCKPS0 = 1;

PR2 = 16; // set PR2 register to 31 - creates interrupt every ~2 ms

/\* formula is Fosc/2 (4MHz) / (prescaler (256) \* PR2 (16) \* 2) = timer frequency

1/timer frequency = timer period \*/

T2CONbits.TSIDL = 0; // continue in idle mode

T2CONbits.T32 = 0; // act as 16 bit timer

T2CONbits.TCS = 0; // use internal clock (Fosc/2)

T2CONbits.TGATE = 0; // disable gated time accumulation

t2intflag = 0; // clear the interrupt flag status

TMR2 = 0; // clear timer register

T2CONbits.TON = 1; // start timer

}

void initializeInterrupts(){

INTCON1bits.NSTDIS = 1; // nested interrupts are not desired

IEC0bits.T2IE = 1; // enable interrupt for Timer 2

IPC1bits.T2IP = 0b111; // set priority to highest (7)

t2intflag = 0; // clear the interrupt flag status

IEC1bits.SI2C1IE = 1; // enable interrupt

}

void sendBytes(unsigned long long playThis){

sendByte1 = playThis>>56 & 0xFF;

sendByte2 = (0xFF & (playThis>>48));

sendByte3 = (0xFF & (playThis>>40));

sendByte4 = (0xFF & (playThis>>32));

sendByte5 = (0xFF & (playThis>>24));

}

void updateEffectsButtons(uint8\_t output[]){

unsigned long long temp = 1;

if(output[0]==1 && flag2[0]==0 && LEDon[0] == 0){EF1\_L=0; flag2[0]=1; LEDon[0] = 1;

LEDon[1] = 0; EF2\_L=1; LEDon[2] = 0; EF3\_L=1; LEDon[3] = 0; EF4\_L=1; LEDon[4] = 0; EF5\_L=1; LEDon[5] = 0; EF6\_L=1;

playThis = (playThis | (temp<<27));} // turn LED on and the mode

else if (output[0]==1 && flag2[0]==0 && LEDon[0] == 1){EF1\_L=0; flag2[0]=1; // turn LED off

LEDon[0] = 1; playThis = (playThis | (temp<<27)); // turn on 1st LED and go back to standard mode

}

else {flag2[0]=output[0];}

if(output[1]==1 && flag2[1]==0 && LEDon[1] == 0){EF2\_L=0; flag2[1]=1; LEDon[1] = 1;

LEDon[0] = 0; EF1\_L=1; LEDon[2] = 0; EF3\_L=1; LEDon[3] = 0; EF4\_L=1; LEDon[4] = 0; EF5\_L=1; LEDon[5] = 0; EF6\_L=1;

playThis = (playThis | (temp<<26));} // turn LED on

else if (output[1]==1 && flag2[1]==0 && LEDon[1] == 1){EF2\_L=1; flag2[1]=1; LEDon[1] = 0;// turn LED off

LEDon[0] = 1; EF1\_L=0; playThis = (playThis | (temp<<27)); // turn on 1st LED and go back to standard mode

}

else {flag2[1]=output[1];}

if(output[2]==1 && flag2[2]==0 && LEDon[2] == 0){EF3\_L=0; flag2[2]=1; LEDon[2] = 1;

LEDon[0] = 0; EF1\_L=1; LEDon[1] = 0; EF2\_L=1; LEDon[3] = 0; EF4\_L=1; LEDon[4] = 0; EF5\_L=1; LEDon[5] = 0; EF6\_L=1;

playThis = (playThis | (temp<<25));} // turn LED on

else if (output[2]==1 && flag2[2]==0 && LEDon[2] == 1){EF3\_L=1; flag2[2]=1; LEDon[2] = 0; // turn LED off

LEDon[0] = 1; EF1\_L=0; playThis = (playThis | (temp<<27)); // turn on 1st LED and go back to standard mode

}

else {flag2[2]=output[2];}

if(output[3]==1 && flag2[3]==0 && LEDon[3] == 0){EF4\_L=0; flag2[3]=1; LEDon[3] = 1;

LEDon[0] = 0; EF1\_L=1; LEDon[1] = 0; EF2\_L=1; LEDon[2] = 0; EF3\_L=1; LEDon[4] = 0; EF5\_L=1; LEDon[5] = 0; EF6\_L=1;

playThis = (playThis | (temp<<27));} // turn LED on

else if (output[3]==1 && flag2[3]==0 && LEDon[3] == 1){EF4\_L=1; flag2[3]=1; LEDon[3] = 0; // turn LED off

LEDon[0] = 1; EF1\_L=0; playThis = (playThis | (temp<<27)); // turn on 1st LED and go back to standard mode

} // turn LED off

else {flag2[3]=output[3];}

if(output[4]==1 && flag2[4]==0 && LEDon[4] == 0){EF5\_L=0; flag2[4]=1; LEDon[4] = 1;

LEDon[0] = 0; EF1\_L=1; LEDon[1] = 0; EF2\_L=1; LEDon[2] = 0; EF3\_L=1; LEDon[3] = 0; EF4\_L=1; LEDon[5] = 0; EF6\_L=1;

playThis = (playThis | (temp<<27));} // turn LED on

else if (output[4]==1 && flag2[4]==0 && LEDon[4] == 1){EF5\_L=1; flag2[4]=1; LEDon[4] = 0; // turn LED off

LEDon[0] = 1; EF1\_L=0; playThis = (playThis | (temp<<27)); // turn on 1st LED and go back to standard mode

}

else {flag2[4]=output[4];}

if(output[5]==1 && flag2[5]==0 && LEDon[5] == 0){EF6\_L=0; flag2[5]=1; LEDon[5] = 1;

LEDon[0] = 0; EF1\_L=1; LEDon[1] = 0; EF2\_L=1; LEDon[2] = 0; EF3\_L=1; LEDon[3] = 0; EF4\_L=1; LEDon[4] = 0; EF5\_L=1;

playThis = (playThis | (temp<<27));} // turn LED on

else if (output[5]==1 && flag2[5]==0 && LEDon[5] == 1){EF6\_L=1; flag2[5]=1; LEDon[5] = 0; // turn LED off

LEDon[0] = 1; EF1\_L=0; playThis = (playThis | (temp<<27)); // turn on 1st LED and go back to standard mode

}

else {flag2[5]=output[5];}

if(output[6]==1 && flag2[6]==0 && LEDon[6] == 0){EF7\_L=0; flag2[6]=1; LEDon[6] = 1;} // turn LED on

else if (output[6]==1 && flag2[6]==0 && LEDon[6] == 1){EF7\_L=1; flag2[6]=1; LEDon[6] = 0;} // turn LED off

else {flag2[6]=output[6];}

if(output[7]==1 && flag2[7]==0 && LEDon[7] == 0){EF8\_L=0; flag2[7]=1; LEDon[7] = 1;} // turn LED on

else if (output[7]==1 && flag2[7]==0 && LEDon[7] == 1){EF8\_L=1; flag2[7]=1; LEDon[7] = 0;} // turn LED off

else {flag2[7]=output[7];}

if(LEDon[0]==1){playThis = (playThis | (temp<<27));}

else if(LEDon[1]==1){playThis = (playThis | (temp<<26));}

else if(LEDon[2]==1){playThis = (playThis | (temp<<25));}

else if(LEDon[3]==1){playThis = (playThis | (temp<<27));}

else if(LEDon[4]==1){playThis = (playThis | (temp<<27));}

else if(LEDon[5]==1){playThis = (playThis | (temp<<27));}

}

void updateQuadEncoder(uint8\_t output[]){

if(output[A]==1 && oldA==0 && output[B]==0){

if(key==11){key=0;}

else key++;

}

else if(output[A]==1 && oldA==0 && output[B]==1){

if(key==0){key=11;}

else key--;

}

else {}

oldA=output[A];

switch (key){

case 0:

Display\_C();

break;

case 1:

Display\_C\_Sharp();

break;

case 2:

Display\_D();

break;

case 3:

Display\_D\_Sharp();

break;

case 4:

Display\_E();

break;

case 5:

Display\_F();

break;

case 6:

Display\_F\_Sharp();

break;

case 7:

Display\_G();

break;

case 8:

Display\_G\_Sharp();

break;

case 9:

Display\_A();

break;

case 10:

Display\_A\_Sharp();

break;

case 11:

Display\_B();

break;

default:

Display\_All();

break;

}

}

void updateStates(uint8\_t output[]){

updateQuadEncoder(output);

updateChordButtons(output);

updatePianoKeys(output);

updateEffectsButtons(output);

/\*

if(output[1]==1){EF2\_L=0;} // turn LED on

else EF2\_L=1; // turn LED off

if(output[2]==1){EF3\_L=0;} // turn LED on

else EF3\_L=1; // turn LED off

if(output[3]==1){EF4\_L=0;} // turn LED on

else EF4\_L=1; // turn LED off

if(output[4]==1){EF5\_L=0;} // turn LED on

else EF5\_L=1; // turn LED off

if(output[5]==1){EF6\_L=0;} // turn LED on

else EF6\_L=1; // turn LED off

if(output[6]==1){EF7\_L=0;} // turn LED on

else EF7\_L=1; // turn LED off

if(output[7]==1){EF8\_L=0;} // turn LED on

else EF8\_L=1; // turn LED off

//if(output[8]==1){DOon}

//else DOoff;

\*/

}

uint8\_t number;

void debounce\_switches(void){

for (number=0; number<numPins; number++) { // go through all the pitches/pins

// Goal: We want 0.25\*newvalue + 0.75\*oldvalue of pin

temp = (oldvalue[number] >> 2); // divide oldvalue by 4

oldvalue[number] = oldvalue[number] - temp; // create 0.75\*oldvalue by subtraction

/\* Check respective pin, add .25 of newvalue to oldvalue \*/

switch (number){

case 0:

if(EF1\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;} // 0x3F is about 0.25 of a uint8

break;

case 1:

if(EF2\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 2:

if(EF3\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 3:

if(EF4\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 4:

if(EF5\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 5:

if(EF6\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 6:

if(EF7\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 7:

if(EF8\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 8:

if(QUAD\_SIG\_A==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 9:

if(QUAD\_SIG\_B==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 10:

if(PIANO1==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 11:

if(PIANO2==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 12:

if(PIANO3==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 13:

if(PIANO4==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 14:

if(PIANO5==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 15:

if(PIANO6==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 16:

if(PIANO7==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 17:

if(PIANO8==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 18:

if(PIANO9==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 19:

if(PIANO10==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 20:

if(PIANO11==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 21:

if(PIANO12==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 22:

if(PIANO13==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 23:

if(PIANO14==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 24:

if(PIANO15==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 25:

if(PIANO16==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 26:

if(PIANO17==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 27:

if(PIANO18==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 28:

if(PIANO19==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 29:

if(PIANO20==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 30:

if(PIANO21==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 31:

if(PIANO22==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 32:

if(PIANO23==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 33:

if(PIANO24==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 34:

if(PIANO25==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 35:

if(CHORD1==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 36:

if(CHORD2==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 37:

if(CHORD3==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 38:

if(CHORD4==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 39:

if(CHORD5==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 40:

if(CHORD6==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 41:

if(CHORD7==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 42:

if(MODE\_1==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

case 43:

if(MODE\_2==buttonPress){oldvalue[number] = oldvalue[number]+0x3F;}

break;

default:

break;

}

/\* Schmidt trigger - if 0.25\*newvalue + 0.75\*oldvalue is above or below 0.5\*/

if((oldvalue[number] > 0xF0)&&(flag[number]==0)){flag[number]=1; output[number]=1;}

if((oldvalue[number] < 0x0F)&&(flag[number]==1)){flag[number]=0; output[number]=0;}

}

updateStates(output); // change outputs according to whether the button is pushed or not

}

#define CHORDI 0x8900000000000000

#define CHORDII 0x2440000000000000

#define CHORDIII 0x0910000000000000

#define CHORDIV 0x0448000000000000

#define CHORDV 0x0112000000000000

#define CHORDVI 0x0048800000000000

#define CHORDVII 0x0012400000000000

void updateChordButtons(uint8\_t output[]){

unsigned long long chord = 0;

playThis = 0x0000000000000000; // initialize it

if(output[35]==1) // if the chord button I is pressed

{chord = CHORDI;}

else if(output[36]==1) // if the chord button II is pressed

{chord = CHORDII;}

else if(output[37]==1){ // if the chord button III is pressed

chord = CHORDIII;}

else if(output[38]==1){ // if the chord button III is pressed

chord = CHORDIV;}

else if(output[39]==1){ // if the chord button III is pressed

chord = CHORDV;}

else if(output[40]==1){ // if the chord button III is pressed

chord = CHORDVI;}

else if(output[41]==1){ // if the chord button III is pressed

chord = CHORDVII;}

playThis = (playThis | (chord>>key)); // add chord frequencies

}

#define CdontPlay 0xAD5AD5FFFFFFFFFF

void updatePianoKeys(uint8\_t output[]){

uint8\_t counter;

long long temp;

long long dontPlay = CdontPlay;

if(output[42]==1) // if STANDARD MODE (play keyboard as normal) (MODE\_1 == 1)

{

for (counter=0; counter<25; counter++) {

temp = output[10+counter]; // need a long long variable if we're going to shift it

playThis = (playThis | (temp<<(63-counter))); // put button info into a long long variable "playThis"

}

}

else{ // ASSIST MODE (move the data around so that the keys being played correspond to the right key

for (counter=0; counter<25; counter++) {

temp = output[10+counter]; // go through the piano buttons on the board

playThis = (playThis | (temp<<(63-key-counter)));

}

// get rid of notes that shouldn't be played (0 = don't play, 1 = don't care)

dontPlay = dontPlay>>key; // shift the keyboard over to make "key" be C

playThis = (playThis & dontPlay);

}

}

void \_\_attribute\_\_((\_\_interrupt\_\_, \_\_shadow\_\_)) \_T2Interrupt(void)

{

debounce\_switches();

t2intflag = 0; //Reset Timer2 interrupt flag and Return from ISR

}

void initializeI2C(void){

I2C1BRG = 156; //132 // I21BRG = (1/I2C\_CLK - 130ns)\*SYS\_CLK-2

// made clock 400 kHz // 156 makes it 100 kHz

I2C1CONbits.I2CSIDL = 0; // operate in idle mode

I2C1CONbits.IPMIEN = 0; // ?????

I2C1CONbits.A10M = 0; // 7 bit slave address

I2C1CONbits.DISSLW = 1; // disable slew rate control

I2C1CONbits.SMEN = 1; // enable pin thresholds

I2C1CONbits.STREN = 0; // disable clock stretching

I2C1CONbits.ACKDT = 1; // send NACK during Acknowledge

I2C1CONbits.I2CEN = 1; // enable I2C module

}

unsigned char buttonsToByte(char B1, char B2, char B3, char B4, char B5, char B6, char B7, char B8){

unsigned char buttonByte=0x00;

buttonByte = B1<<7;

buttonByte = (buttonByte | (B2<<6));

buttonByte = (buttonByte | (B3<<5));

buttonByte = (buttonByte | (B4<<4));

buttonByte = (buttonByte | (B5<<3));

buttonByte = (buttonByte | (B6<<2));

buttonByte = (buttonByte | (B7<<1));

buttonByte = (buttonByte | (B8));

return buttonByte;

}

int main(int argc, char\*\* argv) {

initiateClock();

initiateTimer();

initializeInterrupts();

initialize\_tris();

initializeI2C();

LEDon[0]=1;

EF1\_L=0;

//Initialize Tris

TRISAbits.TRISA15 = 1;

TRISAbits.TRISA5 = 1;

TRISAbits.TRISA4 = 1;

TRISAbits.TRISA14 = 1;

// PORT C

TRISCbits.TRISC13 = 1;

TRISCbits.TRISC15 = 1;

TRISCbits.TRISC12 = 1; //

TRISCbits.TRISC14 = 1;

// PORT D

TRISDbits.TRISD1 = 1;

TRISDbits.TRISD3 = 1;

TRISDbits.TRISD9 = 1;

TRISDbits.TRISD11 = 1;

TRISDbits.TRISD0 = 1;

TRISDbits.TRISD2 = 1;

TRISDbits.TRISD8 = 1;

TRISDbits.TRISD10 = 1;

/\*Listen Button\*/

TRISDbits.TRISD7 = 1;

TRISDbits.TRISD6 = 1;

/\*Quadrature Encoder\*/

TRISDbits.TRISD4 = 1;

TRISDbits.TRISD5 = 1;

/\*Mode Switch\*/

TRISDbits.TRISD13 = 1;

TRISDbits.TRISD12 = 1;

/\*Piano Buttons\*/

AD1PCFG = 0xFFFF; //Set B Register to Digital

TRISAbits.TRISA0 = 1;

TRISEbits.TRISE8 = 1;

TRISEbits.TRISE9 = 1;

TRISBbits.TRISB5 = 1;

TRISBbits.TRISB4 = 1;

TRISBbits.TRISB3 = 1;

TRISAbits.TRISA9 = 1;

TRISAbits.TRISA10 = 1;

TRISBbits.TRISB8 = 1;

TRISBbits.TRISB9 = 1;

TRISBbits.TRISB10 = 1;

TRISBbits.TRISB11 = 1;

TRISAbits.TRISA1 = 1;

TRISFbits.TRISF13 = 1;

TRISFbits.TRISF12 = 1;

TRISBbits.TRISB12 = 1;

TRISBbits.TRISB13 = 1;

TRISBbits.TRISB14 = 1;

TRISBbits.TRISB15 = 1;

TRISDbits.TRISD14 = 1;

TRISDbits.TRISD15 = 1;

TRISFbits.TRISF4 = 1;

TRISFbits.TRISF5 = 1;

TRISFbits.TRISF3 = 1;

TRISFbits.TRISF2 = 1;

/\*Chord Buttons\*/

TRISCbits.TRISC1 = 1;

TRISCbits.TRISC2 = 1;

TRISCbits.TRISC3 = 1;

TRISCbits.TRISC4 = 1;

TRISGbits.TRISG6 = 1;

TRISGbits.TRISG7 = 1;

TRISGbits.TRISG8 = 1;

/\* 16 Segment LED\*/

// PORT A

TRISAbits.TRISA6 = 1;

TRISAbits.TRISA7 = 1;

// PORT E

TRISEbits.TRISE0 = 1;

TRISEbits.TRISE1 = 1;

TRISEbits.TRISE2 = 1;

TRISEbits.TRISE3 = 1;

TRISEbits.TRISE4 = 1;

TRISEbits.TRISE5 = 1;

TRISEbits.TRISE6 = 1;

TRISEbits.TRISE7 = 1;

// PORT F

TRISFbits.TRISF1 = 1;

// PORT G

TRISGbits.TRISG0 = 1;

TRISGbits.TRISG1 = 1;

TRISGbits.TRISG12 = 1;

TRISGbits.TRISG13 = 1;

TRISGbits.TRISG14 = 1;

TRISGbits.TRISG15 = 1;

char t;

while(1){

sendBytes(playThis);

//sendByte2=0b00010000;

StartI2C1();

IdleI2C1();

t = MasterputcI2C1(0b10101010); //Address Byte

IdleI2C1();

t = MasterputcI2C1(sendByte1); //Send First Byte (C1:G1)

IdleI2C1();

t = MasterputcI2C1(sendByte2); //Send Second Byte (G#1:D#2)

IdleI2C1();

t = MasterputcI2C1(sendByte3); //Send Third Byte (E2:B3)

IdleI2C1();

t = MasterputcI2C1(sendByte4); //Send Fourth Byte (C3:G3)

IdleI2C1();

t = MasterputcI2C1(sendByte5); //Send FIfth Byte (G#3:B4 and EF1:EF4)

IdleI2C1();

StopI2C1();

IdleI2C1();

}

return (EXIT\_SUCCESS);

}

**Header files:**

/\*

\* File: ChordButtons.h

\* Author: Nik

\*

\* Created on April 19, 2013, 3:05 PM

\*/

#ifndef CHORDBUTTONS\_H

#define CHORDBUTTONS\_H

#define CHORD1 PORTGbits.RG8

#define CHORD2 PORTGbits.RG6

#define CHORD3 PORTCbits.RC3

#define CHORD4 PORTGbits.RG7

#define CHORD5 PORTCbits.RC4

#define CHORD6 PORTCbits.RC2

#define CHORD7 PORTCbits.RC1

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef \_\_cplusplus

}

#endif

#endif /\* CHORDBUTTONS\_H \*/

/\*

\* File: EffectsButtons.h

\* Author: Nik

\*

\* Created on April 19, 2013, 10:53 AM

\*/

#ifndef EFFECTSBUTTONS\_H

#define EFFECTSBUTTONS\_H

#define EF1\_L TRISDbits.TRISD3

#define EF2\_L TRISDbits.TRISD1

#define EF3\_L TRISCbits.TRISC13

#define EF4\_L TRISDbits.TRISD11

#define EF5\_L TRISDbits.TRISD9

#define EF6\_L TRISAbits.TRISA15

#define EF7\_L TRISCbits.TRISC15

#define EF8\_L TRISAbits.TRISA5

#define EF1\_B PORTDbits.RD2

#define EF2\_B PORTCbits.RC14

#define EF3\_B PORTDbits.RD0

#define EF4\_B PORTDbits.RD10

#define EF5\_B PORTDbits.RD8

#define EF6\_B PORTAbits.RA14

#define EF7\_B PORTCbits.RC12

#define EF8\_B PORTAbits.RA4

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef \_\_cplusplus

}

#endif

#endif /\* EFFECTSBUTTONS\_H \*/

/\*

\* File: MiscLabels.h

\* Author: Nik

\*

\* Created on April 19, 2013, 3:47 PM

\*/

#ifndef MISCLABELS\_H

#define MISCLABELS\_H

#define LISTEN\_L PORTDbits.RD7

#define LISTEN\_B PORTDbits.RD6

#define QUAD\_SIG\_A PORTDbits.RD4

#define QUAD\_SIG\_B PORTDbits.RD5

#define MODE\_1 PORTDbits.RD13

#define MODE\_2 PORTDbits.RD12

#define I2C1\_SCL1 PORTGbits.RG2

#define I2C1\_SDA1 PORTGbits.RG3

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef \_\_cplusplus

}

#endif

#endif /\* MISCLABELS\_H \*/

/\*

\* File: PianoButtons.h

\* Author: Nik

\*

\* Created on April 19, 2013, 11:50 AM

\*/

#ifndef PIANOBUTTONS\_H

#define PIANOBUTTONS\_H

#define PIANO1 PORTEbits.RE8

#define PIANO2 PORTBbits.RB5

#define PIANO3 PORTBbits.RB3

#define PIANO4 PORTAbits.RA10

#define PIANO5 PORTBbits.RB9

#define PIANO6 PORTBbits.RB11

#define PIANO7 PORTFbits.RF13

#define PIANO8 PORTBbits.RB12

#define PIANO9 PORTBbits.RB14

#define PIANO10 PORTDbits.RD14

#define PIANO11 PORTFbits.RF4

#define PIANO12 PORTFbits.RF3

#define PIANO13 PORTAbits.RA0

#define PIANO14 PORTBbits.RB4

#define PIANO15 PORTEbits.RE9

#define PIANO16 PORTAbits.RA9

#define PIANO17 PORTBbits.RB8

#define PIANO18 PORTBbits.RB10

#define PIANO19 PORTAbits.RA1

#define PIANO20 PORTFbits.RF5

#define PIANO21 PORTFbits.RF2

#define PIANO22 PORTDbits.RD15

#define PIANO23 PORTBbits.RB15

#define PIANO24 PORTBbits.RB13

#define PIANO25 PORTFbits.RF12

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef \_\_cplusplus

}

#endif

#endif /\* PIANOBUTTONS\_H \*/

/\*

\* File: SixteenSegmentLED.h

\* Author: Nik

\*

\* Created on April 20, 2013, 10:49 AM

\*/

#ifndef SIXTEENSEGMENTLED\_H

#define SIXTEENSEGMENTLED\_H

#define P1\_B TRISGbits.TRISG12

#define P2\_A TRISGbits.TRISG13

#define P3\_M TRISEbits.TRISE2

#define P4\_K TRISEbits.TRISE3

#define P5\_H TRISEbits.TRISE4

#define P6\_G TRISGbits.TRISG15

#define P7\_T TRISEbits.TRISE5

#define P8\_F TRISEbits.TRISE6

#define P9\_E TRISEbits.TRISE7

#define P10\_DP TRISGbits.TRISG14

#define P11\_S TRISEbits.TRISE1

#define P12\_R TRISEbits.TRISE0

#define P13\_D TRISAbits.TRISA7

#define P14\_U TRISAbits.TRISA6

#define P15\_P TRISGbits.TRISG0

#define P16\_C TRISGbits.TRISG1

#define P17\_N TRISFbits.TRISF1

#ifdef \_\_cplusplus

extern "C" {

#endif

#ifdef \_\_cplusplus

}

#endif

#endif /\* SIXTEENSEGMENTLED\_H \*/

/\*

\* File: 16SegmentLED.c

\* Author: Zach

\*

\* Created on April 19, 2013, 4:01 AM

\*/

#include <stdio.h>

#include <stdlib.h>

#include <p24FJ128GA010.h>

#include "SixteenSegmentLED.h"

void Display\_A(){

P2\_A=0;

P1\_B=0;

P16\_C=0;

P13\_D=0;

P9\_E=1;

P8\_F=1;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=0;

P15\_P=0;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=1;

}

void Display\_B(){

P2\_A=0;

P1\_B=0;

P16\_C=0;

P13\_D=0;

P9\_E=0;

P8\_F=0;

P6\_G=1;

P5\_H=1;

P4\_K=0;

P3\_M=1;

P17\_N=1;

P14\_U=1;

P15\_P=0;

P7\_T=0;

P11\_S=1;

P12\_R=1;

P10\_DP=1;

}

void Display\_C(){

P2\_A=0;

P1\_B=0;

P16\_C=1;

P13\_D=1;

P9\_E=0;

P8\_F=0;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=1;

P15\_P=1;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=1;

}

void Display\_D(){

P2\_A=0;

P1\_B=0;

P16\_C=0;

P13\_D=0;

P9\_E=0;

P8\_F=0;

P6\_G=1;

P5\_H=1;

P4\_K=0;

P3\_M=1;

P17\_N=1;

P14\_U=1;

P15\_P=1;

P7\_T=0;

P11\_S=1;

P12\_R=1;

P10\_DP=1;

}

void Display\_E(){

P2\_A=0;

P1\_B=0;

P16\_C=1;

P13\_D=1;

P9\_E=0;

P8\_F=0;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=0;

P15\_P=1;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=1;

}

void Display\_F(){

P2\_A=0;

P1\_B=0;

P16\_C=1;

P13\_D=1;

P9\_E=1;

P8\_F=1;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=0;

P15\_P=1;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=1;

}

void Display\_G(){

P2\_A=0;

P1\_B=0;

P16\_C=1;

P13\_D=0;

P9\_E=0;

P8\_F=0;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=1;

P15\_P=0;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=1;

}

void Display\_A\_Sharp(){

P2\_A=0;

P1\_B=0;

P16\_C=0;

P13\_D=0;

P9\_E=1;

P8\_F=1;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=0;

P15\_P=0;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=0;

}

void Display\_All(){

P2\_A=0;

P1\_B=0;

P16\_C=0;

P13\_D=0;

P9\_E=0;

P8\_F=0;

P6\_G=0;

P5\_H=0;

P4\_K=0;

P3\_M=0;

P17\_N=0;

P14\_U=0;

P15\_P=0;

P7\_T=0;

P11\_S=0;

P12\_R=0;

P10\_DP=0;

}

void Display\_C\_Sharp(){

P2\_A=0;

P1\_B=0;

P16\_C=1;

P13\_D=1;

P9\_E=0;

P8\_F=0;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=1;

P15\_P=1;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=0;

}

void Display\_D\_Sharp(){

P2\_A=0;

P1\_B=0;

P16\_C=0;

P13\_D=0;

P9\_E=0;

P8\_F=0;

P6\_G=1;

P5\_H=1;

P4\_K=0;

P3\_M=1;

P17\_N=1;

P14\_U=1;

P15\_P=1;

P7\_T=0;

P11\_S=1;

P12\_R=1;

P10\_DP=0;

}

void Display\_F\_Sharp(){

P2\_A=0;

P1\_B=0;

P16\_C=1;

P13\_D=1;

P9\_E=1;

P8\_F=1;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=0;

P15\_P=1;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=0;

}

void Display\_G\_Sharp(){

P2\_A=0;

P1\_B=0;

P16\_C=1;

P13\_D=0;

P9\_E=0;

P8\_F=0;

P6\_G=0;

P5\_H=0;

P4\_K=1;

P3\_M=1;

P17\_N=1;

P14\_U=1;

P15\_P=0;

P7\_T=1;

P11\_S=1;

P12\_R=1;

P10\_DP=0;

}

**Relevant Data Sheets:**

|  |  |
| --- | --- |
| **Part** | **Data sheet** |
| PIC24 | <http://ww1.microchip.com/downloads/en/DeviceDoc/39747F.pdf> |
| dsPIC | <http://ww1.microchip.com/downloads/en/DeviceDoc/70291G.pdf> |
| memory | <http://ww1.microchip.com/downloads/en/DeviceDoc/25071A.pdf> |
| Speaker | <http://www.puiaudio.com/pdf/AS07104PO-WR-R.pdf> |
| Mic | <http://www.cui.com/Product/Resource/DigiKeyPDF/CMA-4544PF-W.pdf> |
| 16-segment | <http://www.digikey.com/product-detail/en/LTP-587G/160-1106-ND/153554> |
| Audio Amp | <http://www.ti.com/lit/ds/symlink/tpa0233.pdf> |
| Dual OpAmp | <http://www.ti.com/lit/ds/slos481a/slos481a.pdf> |
| Quad OpAmp | <http://ww1.microchip.com/downloads/en/DeviceDoc/21685d.pdf> |