Sound generator using PIC16F628A

Elias Riedel Gårding
NMA11, Norra Real, Stockholm

January 15, 2014

Contents

1 Overview

1.1 Objective

The goal of this project was to construct a prototype of a sound generator using Microchip's PIC16F628A microprocessor. It was carried out as an examination project in the course *Tillämpad digitalteknik med PIC-processor* `IL131V` under the supervision of William Sandqvist of KTH, Stockholm.

1.2 Operation

The user selects a tone by turning a rotary encoder, and is given visual feedback from an LCD (Liquid Crystal Display). They may then press and hold a button to hear the tone played, in the form of a square wave, from a speaker. In addition, the user may adjust the speaker's volume by turning a potentiometer. The original intent was to include a possibility for the user to select the waveform of the sound (square, sawtooth or sine wave), but this was dropped due to time constraints. The available tones range from A_4 (440 Hz) to $G_7\#$ (approximately 3322 Hz).

1.3 Functional blocks

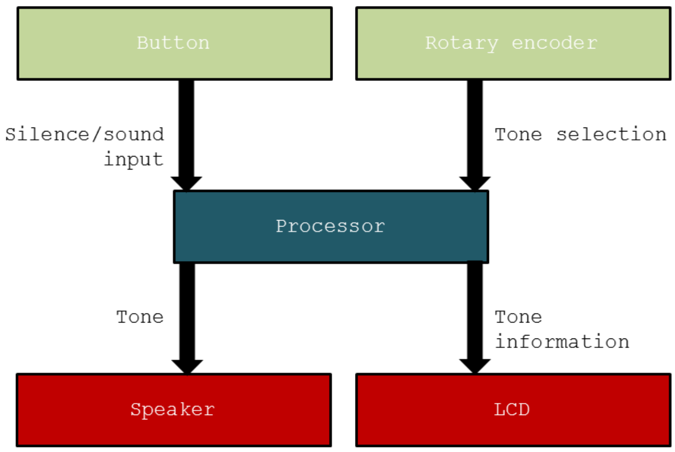

Figure 1: Block diagram.

Figure 1 shows a block diagram of the device's main features. We see that the processor has four main tasks, one for each block:

- Accept and process input from the button
- Accept and process input from the rotary encoder
- Play the desired tone on the speaker
- Display information on the LCD

Button input presented no unexpected problems. As the device employs a press-and-hold control scheme no considerations of switch debouncing were necessary. Input from the rotary encoder was handled using an implementation of a Moore state machine. Sound generation and output to the speaker was handled using PWM (Pulse Width Modulation) from the processor's built-in CCP (Capture, Compare, PWM) unit. Displaying of information on the LCD was achieved by using the LCD's serial/parallel interface.

The datasheet ([2]) was extremely useful in all aspects of the design.

2 Hardware

2.1 Realization

The prototype was realized on a breadboard using components most of which were ordered from Electrokit ([1]). Figure 2 shows the finished prototype.

2.2 Circuit diagram

Figure 3 shows a circuit diagram of the finished prototype. The components shown are only those necessary for the device to function as a sound generator. In addition to these, the breadboard was equipped with components used for in-circuit programming of the processor using Microchip's PICKit 2 programmer.

The button (`BUTTON`) was connected to the `RB2/TX/CK` pin in an active-high fashion using a pull-down resistor (`R1`). The `A` pin of the rotary encoder (`ROTARY_ENCODER`) was connected to the `RB0/INT` pin and the `B` pin was connected to the `RB1/RX/DT` pin. Both were pulled down to ground using a resistor net (`RN1`). The speaker was connected to the `RB3/CCP1` pin via a potentiometer (`VOLUME`). The LCD was connected via six pins to the processor: pins `DB4` through `DB7` were connected to pins `RB4` through `RB7` of the processor, the `RS` pin of the LCD was connected to `RA0/AN0` and its `E`

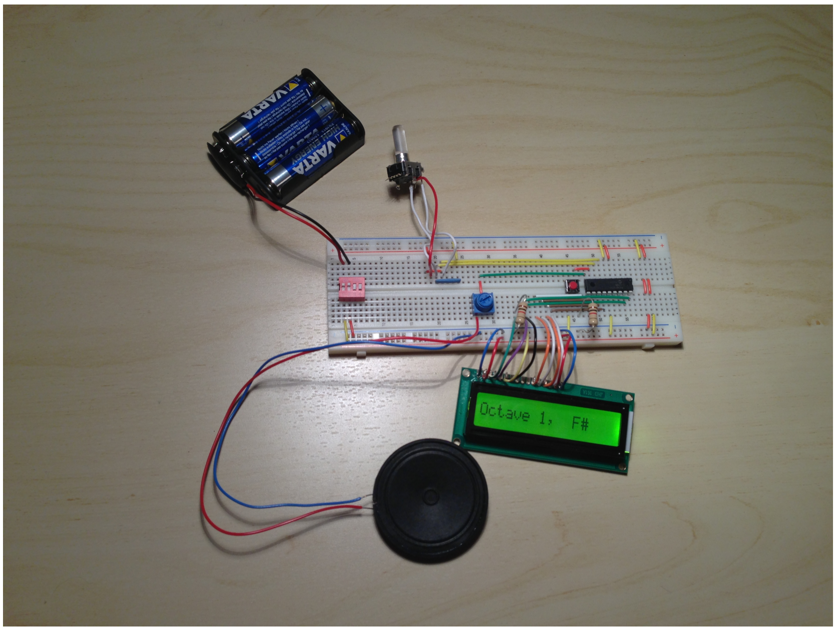

Figure 2: The finished prototype, with components related to programming the processor removed.

pin was connected to `RA1`. The $\texttt{V}_{\texttt{SS}}$ and $\texttt{V}_{\texttt{CC}}$ pins of the LCD were connected to ground and +`5V` respectively, as were the `LED−` and `LED+` pins (`A` and `K` on the circuit diagram). The $\texttt{V}_{\texttt{EE}}$ pin (contrast, `V0` on the diagram) was connected to ground via a resistor (`R2`).

3 Software

3.1 Code and program architecture

Figure 4 and 5 show JSP (Jackson Structured Programming) diagrams of the main program flow; the `main` function and its helper `update_tone`. The functions used as helpers to these functions are better understood by studying their source code, attached in section 5.

The software was written in C and compiled using B Knudsen Data's CC5X C compiler (which uses its own version of C, not ANSI-C). The code was split into six files (see section 5 for listings of their source code):

- `pin_configuration.c`
- `main.c`

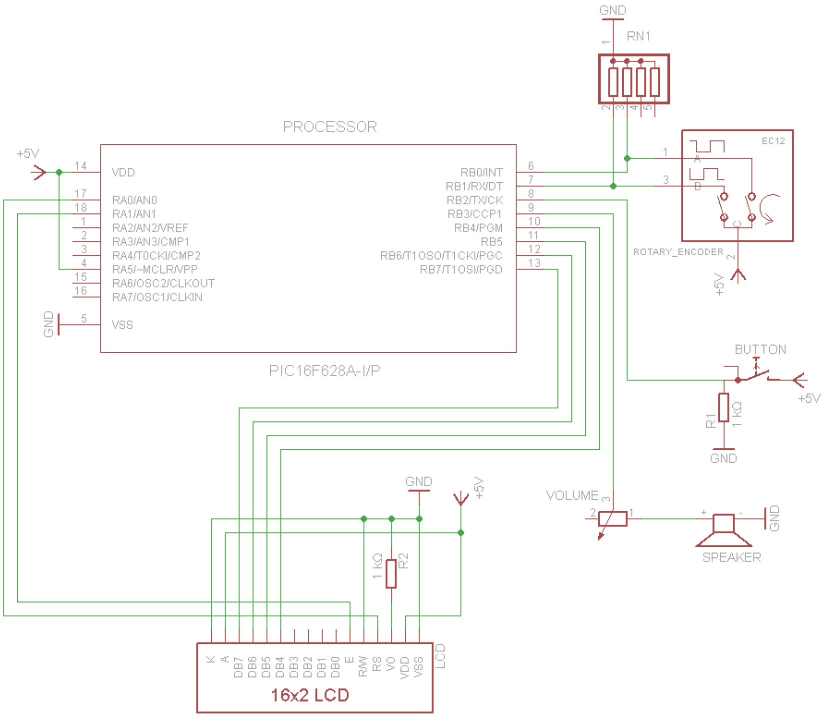

Figure 3: A circuit diagram of the finished prototype.

- `initialization.c`
- `tones.c`
- `lcd.c`
- `delay.c`

`pin_configuration.c` is a pure configuration file, with a graphical overview of which pins are used for what purpose, and mapping of those pins to appropriate aliases.

`main.c` is the main file, it includes every other file and is the file that is fed to the CC5X compiler. It contains the main loop and the code responsible for handling input, including the rotary encoder state machine (see section 3.2). It also contains the preamble of the code, with inclusion of the processor-specific header file, constant definitions, function declarations and so forth.

`initialization.c` handles initial setting of configuration registers such as the `TRIS`-registers and the configuration registers of the CCP unit.

`tones.c` deals with the production of sound on the speaker. It uses the processor's `CCP1` module, set to PWM mode, to produce a square wave current on the `RB3/CCP1` pin. This is elaborated on in section 3.3.

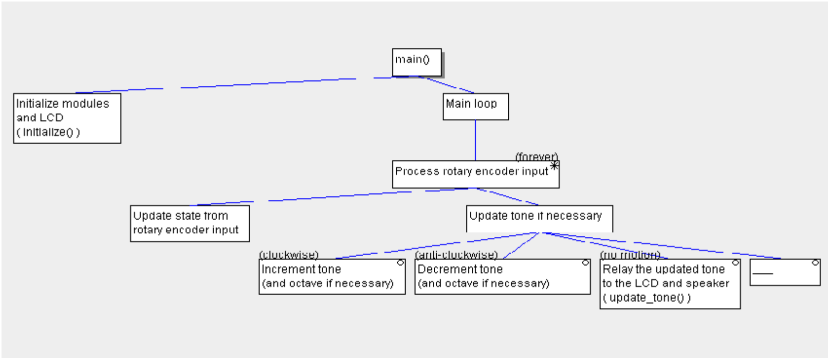

Figure 4: A JSP diagram of the `main` function.

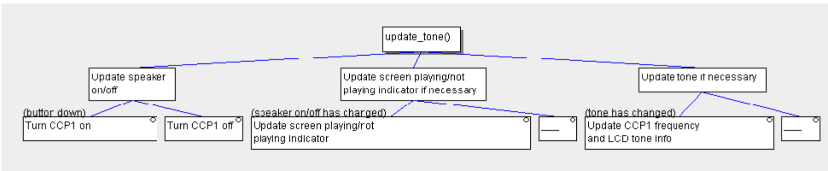

Figure 5: A JSP diagram of the `update_tone` function.

`lcd.c` contains functions involved in interacting with the LCD. Commands are sent to the display in 4-bit mode (to which the display is set during initialization), thus requiring six lines between the display and the processor: The `E` (enable) line, the `RS` (command/character) line, and the four data lines `DB4` through `DB7`.

`delay.c` contains a single function `instr_delay_ms`, used throughout the program to produce a delay in program execution.

3.2 Rotary encoder input

Input from the rotary encoder is handled through a Moore state machine. The concept and implementation are almost exactly copied from a lab in the course ([4]). A state is described by two bits: the values of the A and B pins of the rotary encoder in that state. At each iteration of the main loop (see figure 4 and the listing of `main.c`) the current *state transition* (`rotary_encoder_state_transition`) is calculated by appending the current state to the previous state. If the transition is from state 00 to 01 the tone is incremented. If it is from 01 to 00 the tone is decremented. On a transision from 00 to 00 (no motion of the rotary encoder) the tone and the LCD are updated if necessary.

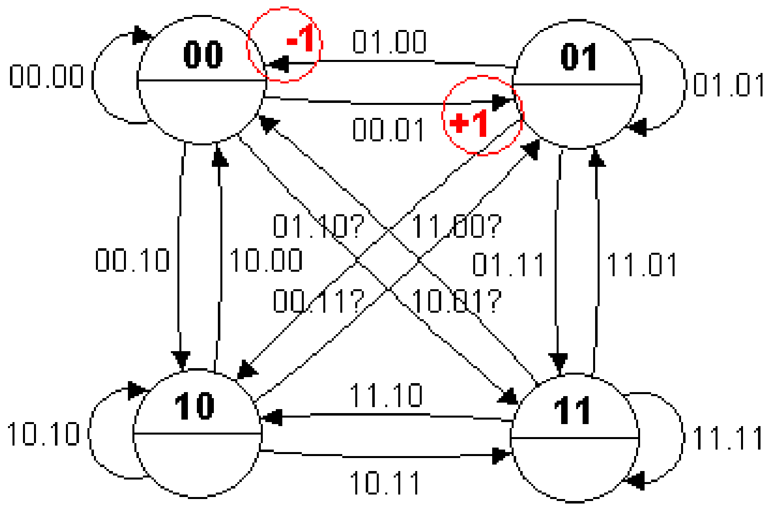

Figure 6: A state diagram of the Moore state machine. (Image from [4])

3.3 Tone generation

Tone generation is done using the processor's built in CCP (Capture, Compare, PWM) unit set to PWM (Pulse Width Modulation) mode. Once the correct settings for a certain tone are in place, the unit will produce an oscillating voltage on the `RB3/CCP1` pin, independently of program execution. Switching the tone on and off is achieved through simply deactivating the CCP unit in its control register.

The frequency of a tone t in octave o was calculated as

$$f = 440 * 2^{o+\frac{t}{12}} \text{ [Hz]}$$

The processor uses two lookup functions (`tone_PR2_value` and `TMR2_prescaler_configuration_bits`, located in `tones.c`) together with some arithmetic to determine the necessary values of the `PR2` and `CCPR1L` registers, as well as the `CCP1X` and `CCP1Y` bits of the `CCP1CON` register and the `T2CKPS1` and `T2CKPS0` bits of the `T2CON` register, for a desired tone.

The values in this lookup table were calculated using a Python program, finding the correct values of the `PR2` register and the two prescaler configuration bits (`T2CKPS1` and `T2CKPS0`) such that the `PR2` value fits within a byte and the prescale ratio is as small as possible. The data sheet gives the formulae

$$T_{\texttt{PWM}} = (\texttt{PR2} + 1) \;\; 4 \; T_{\texttt{OSC}} \; p$$

and

$$C_{\texttt{PWM}} = \texttt{CCPR1}\ T_{\texttt{OSC}}\ p$$

where $T_{\texttt{PWM}}$ is the period of the output signal ($T_{\texttt{PWM}} = 1/f$), PR2 is the value of the PR2 register, $T_{\texttt{OSC}}$ is the oscillator period, p is the Timer2 prescaler ratio, $C_{\texttt{PWM}}$ is the PWM duty cycle and CCPR1 is the 10-bit number given by appending the bits CCP1X and CCP1Y to the CCPR1L register. Since the signal is to become a classical square wave the amount of time spent high should be the same as the amount of time spent low. Therefore

$$T_{\texttt{PWM}} = 2\ C_{\texttt{PWM}}$$

The three equations together give (after simplification) that

$$\texttt{CCPR1} = 2\ \ (\texttt{PR2} + 1)$$

This makes it possible to store the necessary information as just the values PR2 and T2CKPS1 : T2CKPS0 (the prescale ratio), and then calculate CCPR1 from those values.

4 Adequacy tests

4.1 Interface functionality

Check that correct information is always displayed on the LCD, and that a tone is produced when the button is pressed. Check that the rotary encoder switches tones in intervals of one semi-tone every time it gives a pulse, and that the tone wraps around correctly when the maximum or minimum tone is passed.

4.2 Pitch accuracy

Use a reliable source of pitch and compare it to the output of the sound generator, or better yet, use an oscilloscope to measure the frequency of the produced tones.

5 Source code listings

5.1 pin_configuration.c

```
// FILE:  pin_configuration.c

/*
                     --------  --------
                    |        \/        |
                    |RA2   16F628   RA1|--LCD_EN
                    |RA3            RA0|--LCD_RS
                    |RA4-od    RA7/OSC1|
                    |RA5/~MCLR RA6/OSC2|
               GND--|Vss            Vdd|--  +5V
  Rotary encoder A--|RB0/INT  (RB7)/PGD|--LCD_D7
  Rotary encoder B--|RB1/Rx   (RB6)/PGC|--LCD_D6
            Button--|RB2/Tx         RB5|--LCD_D5
           Speaker--|RB3/CCP1 (RB4)/PGM|--LCD_D4
                    |__________________|
*/

#pragma bit button_down @ PORTB.2

#pragma bit speaker @ PORTB.3

#pragma bit rotary_encoder_A @ PORTB.0
#pragma bit rotary_encoder_B @ PORTB.1

#pragma bit LCD_EN @ PORTA.1
#pragma bit LCD_RS @ PORTA.0
#pragma bit LCD_DB4 @ PORTB.4
#pragma bit LCD_DB5 @ PORTB.5
#pragma bit LCD_DB6 @ PORTB.6
#pragma bit LCD_DB7 @ PORTB.7
```

pin_configuration.c

5.2 main.c

```
// FILE:  main.c

#include "16F628.h"

// Configuration
#pragma config |= 0x3f30

// Create an alias for the char datatype, 'byte', to better describe the datatype
#define byte char


// Constant definitions
#define MIN_OCTAVE 0
#define MAX_OCTAVE 2
#include "pin_configuration.c"

// Global variable declarations
byte current_tone;
byte current_octave;
bit playing;      // Whether a tone is currently playing

byte previous_tone;
byte previous_octave;
bit previous_playing;

byte rotary_encoder_state_transition; // The previous state (2 bits) followed by the current state


// Function declarations
// initialization.c
  void initialize();
// tones.c
  byte tone_PR2_value(byte tone, byte octave);
  byte TMR2_prescaler_configuration_bits(byte tone, byte octave);
  void play_tone();
```

```
// lcd.c
  void LCD_init();
  void LCD_putchar(byte data);
  void LCD_write_string(const char *str);
  void LCD_update_tone();
  void LCD_update_playing();
// delay.c
  void instr_delay_ms(byte ms);
// main.c
  void update_tone();
  void main();

// Inclusion of code files
#include "initialization.c"
#include "tones.c"
#include "lcd.c"
#include "delay.c"

void update_tone()
{
  // Poll button input
  previous_playing = playing;
  playing = button_down;
  if (playing) {
    // Set CCP1 to PWM mode (set bits CCP1M3 and CCP1M2)
    CCP1CON |= 0b0000.1100;
  }
  else {
    // Turn off CCP1 (clear bits CCP1M3 and CCP1M2)
    CCP1CON &= 0b1111.0011;
  }

  // Update the screen and CCP unit where necessary
    if (previous_playing != playing)
      LCD_update_playing();

    if (current_tone != previous_tone
      || current_octave != previous_octave)
    {
      play_tone();
      LCD_update_tone();
    }

  previous_tone = current_tone;
  previous_octave = current_octave;
}

void main()
{
  initialize();

  // Initialize variables
  previous_tone = -1;
  previous_octave = -1;
  previous_playing = 0;
  current_tone = 0;
  current_octave = 0;
  playing = 1;
  rotary_encoder_state_transition = 0b00.00;

  // Ensure that the first tone is displayed
  update_tone();

  while (1) {
    // Process rotary encoder input

    // Update the current state
      // The previous state
      rotary_encoder_state_transition.3 = rotary_encoder_state_transition.1;
      rotary_encoder_state_transition.2 = rotary_encoder_state_transition.0;
      // The current state
      rotary_encoder_state_transition.0 = rotary_encoder_A;
      rotary_encoder_state_transition.1 = rotary_encoder_B;

    if (rotary_encoder_state_transition == 0b00.00) {
      // Update
      update_tone();
    }
    else {
      // Check for increment/decrement transitions
      if (rotary_encoder_state_transition == 0b00.01) { // From 00 to 01
        // Increment tone
        current_tone++;
```

```
        if (current_tone == 12) {
          current_tone = 0;
          // Increment octave
          current_octave++;
          if (current_octave == MAX_OCTAVE+1)
            current_octave = MIN_OCTAVE;
        }
      }
      else if (rotary_encoder_state_transition == 0b01.00) { // From 01 to 00
        // Decrement tone
        current_tone--;
        if (current_tone == -1) {
          current_tone = 11;
          // Decrement octave
          current_octave--;
          if (current_octave == MIN_OCTAVE-1)
            current_octave = MAX_OCTAVE;
        }
      }
    }
  }
}
```

main.c

5.3 initialization.c

```
// FILE:  initialization.c

void initialize()
{
  // Disable comparators on RA0 through RA3
  CMCON = 0b00000.111;
  /*
   *  00xxx.xxx Comparator 2 and 1 Output. Read-only bits.
   *  xx00x.xxx Comparator 2 and 1 Output Inversion. Irrelevant.
   *  xxxx0.xxx Comparator Input Switch. Irrelevant. For connecting comparators to different pins.
   *  xxxxx.111 Comparator Mode. 111: Comparators Off.
   */

  // Configure pins for input or output
  TRISA = 0b1111.1100;
  /*
   *  xxxx.xx0x RA1 pin to be used as output to LCD_EN
   *  xxxx.xxx0 RA0 pin to be used as output to LCD_RS
   */
  TRISB = 0b0000.0111;
  /*
   *  0000.xxxx RB4-RB7 to be used as outputs to LCD_DB4-LCD_DB7 respectively.
   *  xxxx.0xxx RB3/CCP1 pin to be used as output to the speaker.
   *  xxxx.x1xx RB2 pin to be used as input from the button
   *  xxxx.xx11 RB1 and RB0 pins to be used as inputs from rotary encoder A and B respectively
   */

  // Configure the CCP1 unit to start turned off (it is turned on when the button is pressed)
  CCP1CON = 0b0000.0000;
  /*
   *  00xx.xxxx Unimplemented.
   *  xx00.xxxx PWM Least Significant bits. The two LSBs of the PWM duty cycle. Subject to change.
   *  xxxx.1100 CCP1 Mode Select. 0000: Capture/Compare/PWM off.
   */

  // Enable the Timer2 module (for use by the CCP1 unit)
  TMR2ON = 1; // In T2CON register

  // Initialize the LED display (see lcd.c)
  LCD_init();
}
```

initialization.c

5.4 tones.c

```c
// FILE: tones.c

byte tone_PR2_value(byte tone, byte octave)
  // Returns the value that PR2 should assume for the given tone in the given octave.
{
  // Computed Goto
  byte index = 12 * octave + tone;
  skip(index);

  /*
   * A table of the values that PR2 should assume. The octaves are stored one after another.
   * The values are calculated according to the formula (given in the documentation)
   *   PR2 = PWM_period / (4*Tosc * TMR2_prescale) - 1
   * The value TMR2_prescale is calculated such that PR2 can be contained within a single byte.
   * TMR2_prescale is given by TMR2_prescaler_configuration_bits(tone, octave)
   */
  #pragma return[] = \
/* Octave 0 */    141  133  126  118  112  105   99   94   88   83   79   74 \
/* Octave 1 */     70   66  252  238  224  212  200  189  178  168  158  149 \
/* Octave 2 */    141  133  126  118  112  105   99   94   88   83   79   74
  /* Tone name:    A    A#   B    C    C#   D    D#   E    F    F#   G    G# */
  /* Tone index:   0    1    2    3    4    5    6    7    8    9    10   11 */
}

byte TMR2_prescaler_configuration_bits(byte tone, byte octave)
  // Returns a byte containing as its last two bits the bits that T2CKPS1:T2CKPS0
  // should assume for the given tone in the given octave.
{
  byte index = 12 * octave + tone;
  if (index >= 12 * 1 + 2) // Boundary at B in octave 1
    return 0b01; // Prescaler ratio 1:4
  else
    return 0b10; // Prescaler ratio 1:16
}

void play_tone()
{
  /*
   * The documentation gives the following formulae:
   *   PR2 = PWM_period / (4*Tosc * TMR2_prescale) - 1
   *   PWM_duty_cycle = CCPR1 * Tosc * TMR2_prescale
   * Since we want a square wave, we need a duty cycle ratio of 50%. Thus:
   *   PWM_duty_cycle = PWM_period / 2
   * These formulas together give
   *   CCPR1 = 2 * (PR2 + 1)
   * We look up the value of PR2 in a table and then calculate CCPR1 using this formula.
   */

  // Look up the period for the given tone
  PR2 = tone_PR2_value(current_tone, current_octave);

  // A 10-bit variable
  long long_PR2_plus_1 = (long) PR2 + 1;
  long CCPR1 = 2 * long_PR2_plus_1;

  // The two least significant bits
  CCP1X = CCPR1.1;
  CCP1Y = CCPR1.0;

  // The eight most significant bits
  CCPR1 >>= 2;
  CCPR1L = CCPR1;


  // Set the Timer2 prescaler ratio
  byte TMR2_prescaler = TMR2_prescaler_configuration_bits(current_tone, current_octave);
  T2CKPS1 = TMR2_prescaler.1;
  T2CKPS0 = TMR2_prescaler.0;
}
```

tones.c

5.5 lcd.c

```c
// FILE: lcd.c

const char LCD_FIRST_ROW[]  = "Octave  ";
```

```c
const char LCD_SECOND_ROW[] = ",       ";
#define LCD_OCTAVE_LOCATION 0b1.0000111
#define LCD_TONE_LETTER_LOCATION 0b1.1000011
#define LCD_TONE_SUFFIX_LOCATION 0b1.1000100
#define LCD_PLAYING_LOCATION 0b1.1000111

#define LCD_PLAYING_CHARACTER_ADDRESS 0
#define LCD_NOT_PLAYING_CHARACTER_ADDRESS ' '

const char TONE_LETTERS[] = "AABCCDDEFFGG";
const char TONE_SUFFIXES[]  = " #  # #  # #";

void LCD_init()
{
  // Give the LCD time to settle down after Vcc
  instr_delay_ms(80);

  // Put the LCD in Command mode
  LCD_RS = 0;

  /*  Enable 4-bit interface by issuing the command
   *    0010xxxx
   *  It will be sent twice due to the nature of LCD_putchar().
   */
  LCD_putchar(0b0010.0010);
  // Hereafter instructions are sent one nibble at a time,
  // so LCD_putchar will send instructions correctly.

  // Set the display to 2-line mode and 5x10 dot format
  LCD_putchar(0b0010.1000);
  /*
   *  001x.xxxx Function set command
   *  xxx0.xxxx 4-bit interface
   *  xxxx.1xxx 2-line mode
   *  xxxx.x0xx 5x10 dot format
   *  xxxx.xx00 Unimplemented
   */

  // Turn the display on, turn cursor and cursor blink off
  LCD_putchar(0b0000.1100);
  /*
   *  0000.1xxx Display On/Off & Cursor command
   *  xxxx.x1xx Display on
   *  xxxx.xx0x Cursor off
   *  xxxx.xxx0 Cursor blink off
   */

  // Clear the display
  LCD_putchar(0b0000.0001);
  /*
   *  0000.0001 Clear Display command
   */

  // Set the character entry mode to increment, no shift
  LCD_putchar(0b0000.0110);
  /*
   *  0000.01xx Character Entry Mode command
   *  xxxx.xx1x Increment
   *  xxxx.xxx0 No shift
   */

  // Define the tone character
    // Select CGRAM address 000000
    LCD_putchar(0b0100.0000);
    /*
     *  01xx.xxxx Set CGRAM Address command
     *  xx00.0000 The CGRAM address
     */

    // Put the LCD in character mode
    LCD_RS = 1;

    // Define the character
    LCD_putchar(0b000.00011); //    ##
    LCD_putchar(0b000.00010); //    #
    LCD_putchar(0b000.00010); //    #
    LCD_putchar(0b000.01110); //  ###
    LCD_putchar(0b000.11110); // ####
    LCD_putchar(0b000.11110); // ####
    LCD_putchar(0b000.01100); //  ##
    LCD_putchar(0b000.00000); //

  // Move the cursor to the first display address
```

```
  LCD_RS = 0; // Command mode
  LCD_putchar(0b1000.0000);
  /*
   *  1xxx.xxxx Set Display Address command
   *  x000.0000 The display address
   */

  // Write the first row
  LCD_RS = 1; // Character mode
  LCD_write_string(LCD_FIRST_ROW);

  // Move the cursor to the first column of the second row
  LCD_RS = 0; // Command mode
  LCD_putchar(0b1100.0000);
  /*
   *  1xxx.xxxx Set Display Address command
   *  x100.0000 The display address
   */

  // Write the second row
  LCD_RS = 1; // Character mode
  LCD_write_string(LCD_SECOND_ROW);
}

void LCD_putchar(byte data)
  // Send the given data to the LCD in 4-bit mode.
{
  // Load the upper nibble of the data into the bus
  LCD_DB7 = data.7;
  LCD_DB6 = data.6;
  LCD_DB5 = data.5;
  LCD_DB4 = data.4;

  // Tick the LCD
  LCD_EN = 0;
  nop();
  LCD_EN = 1;

  // Give the LCD time to receive the data
  instr_delay_ms(2);

  // Load the lower nibble of the data into the bus
  LCD_DB7 = data.3;
  LCD_DB6 = data.2;
  LCD_DB5 = data.1;
  LCD_DB4 = data.0;

  // Tick the LCD
  LCD_EN = 0;
  nop();
  LCD_EN = 1;

  // Give the LCD time to receive the data
  instr_delay_ms(2);
}

void LCD_write_string(const char *str)
{
  LCD_RS = 1;
  byte i;
  for (i = 0; str[i] != '\0'; i++)
    LCD_putchar(str[i]);
}


void LCD_update_tone()
{
  // Octave
  LCD_RS = 0;
  LCD_putchar(LCD_OCTAVE_LOCATION);
  LCD_RS = 1;
  LCD_putchar('0' + current_octave);

  // Tone letter
  LCD_RS = 0;
  LCD_putchar(LCD_TONE_LETTER_LOCATION);
  LCD_RS = 1;
  LCD_putchar(TONE_LETTERS[current_tone]);

  // Tone suffix
  LCD_RS = 0;
  LCD_putchar(LCD_TONE_SUFFIX_LOCATION);
  LCD_RS = 1;
```

```c
  LCD_putchar(TONE_SUFFIXES[current_tone]);
}

void LCD_update_playing()
{
  // Go to the location of the playing/not playing symbol
  LCD_RS = 0;
  LCD_putchar(LCD_PLAYING_LOCATION);

  // Write the correct character
  LCD_RS = 1;
  if (playing)
    LCD_putchar(LCD_PLAYING_CHARACTER_ADDRESS);
  else
    LCD_putchar(LCD_NOT_PLAYING_CHARACTER_ADDRESS);
}
```

lcd.c

5.6 delay.c

```c
// FILE:  delay.c

void instr_delay_ms(byte ms) // Delay the specified number of milliseconds (1 <= ms <= 256). Error: ~0.1%
{
  // The delay in us of each loop = 4 + i * (4 + j * (5) + 3) + 3.
  do { // Sleep 1 ms
    // 4 + 7 * (4 + 27 * (5) + 3) + 3 = 1001 ins.
    char i = 7; // 2 ins
    do {
      char j = 27; // 2 ins
      do {} while (--j); // 3 ins
    } while (--i); // 3 ins
  } while (--ms); // 3 ins
}
```

delay.c

References

[1] http://www.electrokit.com/

[2] Microchip Technology Inc. *PIC16F627A/628A/648A Data Sheet.* http://web.mit.edu/6.115/www/datasheets/16f628.pdf

[3] B Knudsen Data. *CC5X User's Manual.* Version 3.4. http://www.bknd.com/doc/cc5x-34.pdf

[4] Sandqvist, W. *Avläsning av pulsgivare.* http://www.ict.kth.se/courses/IL131V/quad/index.htm