Contents

1 ArduinoStdInt.h

```
/*
  arduinoStdInt.h - Library for the types in <stdint.h> from C

  Created by Jakob Kunzler December 25 2016
*/

#ifndef arduinoStdInt_h
#define arduinoStdInt_h

#include "Arduino.h"

// Creates the std_int types for use in Arduino

typedef unsigned char uint8_t; // 1 byte int
typedef unsigned int uint16_t; // 2 byte int
typedef unsigned long uint32_t; // 4 byte int
typedef unsigned long long uint64_t; // 8 byte int

typedef signed char int8_t; // 1 byte int
typedef signed int int16_t; // 2 byte int
typedef signed long int32_t; // 4 byte int
typedef signed long long int64_t; // 8 byte int

#endif
```

2 buttonHandler.cpp

```
#ifndef buttonHandler_c
#define buttonHandler_c

#include "buttonHandler.h"

// The Number of IO
#define BH_NUMIO 6
// The length of a char
#define BH_CHARLEN 7
// The threshold voltage on the analog pins for a high
#define BH_VTHRESHOLD 100

// INIT FUNCTION
void BH_initAll()
{
        pinMode(PIN_BTN_0,INPUT);
        pinMode(PIN_BTN_1,INPUT);
        pinMode(PIN_BTN_2,INPUT);
        pinMode(PIN_BTN_3,INPUT);
        pinMode(PIN_BTN_4,INPUT);
        pinMode(PIN_BTN_5,INPUT);
        pinMode(PIN_SW_0,INPUT);
}

// Read an analog IO pin against value BH_VTHRESHOLD
uint8_t _BH_readIO(uint8_t pin)
{
  return(analogRead(pin)>BH_VTHRESHOLD);
}


// Read all of the buttons, use a mask to pull IO out
uint8_t BH_readAll()
{
        // Init IO
        uint8_t io = 0;
        // Read Value
        uint8_t read = _BH_readIO(PIN_SW_0);
        // Shift and add
        io = (io<<1)|read;
        // Repeat...
        read = _BH_readIO(PIN_BTN_5);
        io = (io<<1)|read;
        read = _BH_readIO(PIN_BTN_4);
        io = (io<<1)|read;
        read = _BH_readIO(PIN_BTN_3);
        io = (io<<1)|read;
        read = _BH_readIO(PIN_BTN_2);
        io = (io<<1)|read;
        read = _BH_readIO(PIN_BTN_1);
        io = (io<<1)|read;
        read = _BH_readIO(PIN_BTN_0);
        io = (io<<1)|read;
        // Return the value
        return io;
}

// Get a particular IO
uint8_t BH_getAnIO(char type, uint8_t index)
{
  if (type == 'S')
  {
    index = BH_NUMIO + index;
  }

  // Shift to the correct bit
  uint8_t numLeft = BH_CHARLEN-index;
  uint8_t IO = (BH_readAll()<<numLeft);
  uint8_t output = IO>>BH_CHARLEN;
  return output;

}

#endif
```

3 buttonHandler.h

```c
#ifndef buttonHandler_h
#define buttonHandler_h

#include "ArduinoStdInt.h"
#include "pinList.h"
#include "Arduino.h"

// INIT FUNCTION
void BH_initAll();

// Read all of the buttons, use a mask to pull IO out
uint8_t BH_readAll();

// Get a particular IO
uint8_t BH_getAnIO(char type, uint8_t index);


#endif
```

4 buzzerDriver.cpp

```
#ifndef buzzerDriver_c
#define buzzerDriver_c

#include "buzzerDriver.h"

// Beeps quickly
void BZ_beep()
{
  tone(PIN_BUZZER, BZ_BEEP_FREQ_HZ, BZ_BEEP_DURATION_MS);
}

// Simple alarm, drive on or off
void BZ_alarm(bool on)
{

  if (on)
    tone(PIN_BUZZER, BZ_BEEP_FREQ_HZ);
  else
    noTone(PIN_BUZZER);
}


// Plays BZ_SONG continously while driven on at certain rate (use in a tick
    function)

static unsigned long sound_lastTickMS = 0; // Last time the tick was called in
    milliseconds
static unsigned long sound_currentMS = 0; // Current number of milliseconds
    since program start

unsigned int noteNumber = 0; // Iterator in BZ_SONG array
unsigned int noteBeatsLeft = 0; // Counter of number of beats to hold note.
    Counts down to 0.
unsigned int noteFrequency = 0; // Frequency of note to sustain

#define BZ_SONG_NUM_BYTES (sizeof(BZ_SONG)) // Memmory size of the song
#define BZ_SONG_SIZE (BZ_SONG_NUM_BYTES/(sizeof(unsigned int))) // The length of
     the BZ_SONG array
// Adapted for tempo, the quarter note gets the beat when considering tempo
#define BZ_TICKS_PER_BEAT ((GB_INTERUPTS_PER_SECOND*60)/(BZ_SONG_BPM*
    QUARTER_NOTE))

// This function sustains the next note of the BZ_SONG until the amound of beats
     left for that note is zero.
// Each time tickBZ_SONG is called (every beat) the beat count goes down by 1.
// After finishing a note, it advances to the next note.  It does nothing if the
     BZ_SONG is finished
// When BZ_SONG is complete, return true


bool _BZ_tickBZ_SONG(const unsigned int* song)
{
  if (noteNumber == 0) // Check for Initial State
  {
    // Advance in Array
    noteNumber = noteNumber + 2;
    // Find the intial frequency
    noteFrequency = song[noteNumber];
    // Get the initial number of beats
    noteBeatsLeft = song[noteNumber + 1];
    return false; // Not Over
  }
  else if (noteNumber >= BZ_SONG_SIZE) // BZ_SONGOver
  {
    noTone(PIN_BUZZER); // Silence
    return true; // Do nothing
  }
  else // Play Note
  {
    // Decrement the number of beats left
    noteBeatsLeft--;

    if (noteFrequency == 0)
    {
      // Rest Note
      noTone(PIN_BUZZER);

    }
    else
    { // Drive the note
      tone(PIN_BUZZER, noteFrequency);
    }

    // Check to advance
    if (noteBeatsLeft == 0)
    {
      // Advance in Array
      noteNumber = noteNumber + 2;
      // Find the new frequency
      noteFrequency = (BZ_SONG)[noteNumber];
      // Get the new number of beats
      noteBeatsLeft = (BZ_SONG)[(noteNumber + 1)];
    }
    return false; // Not Over
  }

}

// Counts ticks for the tempo
uint8_t BZ_songTickCounter = 0;

// Plays BZ_SONG continously while driven on
void BZ_alarmBZ_SONG(bool on)
{
  if (on)
  {
    BZ_songTickCounter++;
    if (BZ_songTickCounter>BZ_TICKS_PER_BEAT)
    {
      if(_BZ_tickBZ_SONG(BZ_SONG))
      {
        // Reset and repeat
        noteNumber = 0;
        noteBeatsLeft = 0;
        noteFrequency = 0; ;
      }
      BZ_songTickCounter = 0;
    }
  }
  else
    noTone(PIN_BUZZER);
}

// Init buzzer
void BZ_init()
{
  pinMode(PIN_BUZZER, OUTPUT);
  BZ_songTickCounter = 0;
  BZ_beep();
}

#endif
```

5 buzzerDriver.h

```
#ifndef buzzerDriver_h
#define buzzerDriver_h

#include "Arduino.h"
#include "ArduinoStdInt.h"
#include "pinList.h"
#include "globalParameters.h"

#define BZ_BEEP_FREQ_HZ 2000
#define BZ_BEEP_DURATION_MS 100

// Beeps quickly
void BZ_beep();

// Simple alarm, drive on or off
void BZ_alarm(bool on);

// Plays BZ_SONG continously while driven on at certain rate (use in a tick
    function)
void BZ_alarmBZ_SONG(bool on);

// Init the buzzer
void BZ_init();


/// PROGRAM THE MUSIC BELOW


// Notes

#define NOTE_C4 261
#define NOTE_Db4 277
#define NOTE_D4 294
#define NOTE_Eb4 311
#define NOTE_E4 330
#define NOTE_F4 349
#define NOTE_Gb4 370
#define NOTE_G4 392
#define NOTE_Ab4 415
#define NOTE_A4 440
#define NOTE_Bb4 466
#define NOTE_B4 494
#define NOTE_C5 523
#define NOTE_D5 587
#define NOTE_Eb5 622
#define NOTE_E5 659
#define NOTE_F5 698
#define NOTE_Gb5 740
#define NOTE_G5 784
#define NOTE_Ab5 831
#define NOTE_A5 880
#define NOTE_Bb6 932
#define NOTE_B6 988
#define NOTE_C8 1047
#define NOTE_REST 0
```

```
// Durations
#define THIRTYSECONDTH_NOTE 1
#define SIXTEENTH_NOTE 2
#define EIGTH_NOTE 4
#define QUARTER_NOTE 8
#define HALF_NOTE 16
#define WHOLE_NOTE 32
#define BREATH_MARK (THIRTYSECONDTH_NOTE)

/* Note on the durations:
In implementation, the fundamental shortest duration is a 32th note,
and each of the other notes are multiples of the 32th note durations.

The tempo beats per minute are scaled to assume a n/4 time signature, ie)
the quarter note gets the beat.  There are no measures, just notes.

*/

// BZ_SONG Parameters
#define BZ_SONG_BPM 200 // BZ_SONG Tempo, relative to quarter note
// Note: The faster the tick rate, the more accurate the BPM.
// The shortest a 32nd note can be is one tick.

// BZ_SONG
// Title: Emry
const unsigned int BZ_SONG[] =
{ NOTE_REST, BREATH_MARK, // Skip First Note
  NOTE_C4, QUARTER_NOTE,
  NOTE_E4, QUARTER_NOTE,
  NOTE_G4, QUARTER_NOTE,
  NOTE_C5, QUARTER_NOTE,
  NOTE_B4, QUARTER_NOTE,
  NOTE_A4, QUARTER_NOTE,
  NOTE_G4, HALF_NOTE,
  NOTE_REST, BREATH_MARK,
  //
  NOTE_C4, QUARTER_NOTE,
  NOTE_E4, QUARTER_NOTE,
  NOTE_G4, QUARTER_NOTE,
  NOTE_C5, QUARTER_NOTE,
  NOTE_B4, QUARTER_NOTE,
  NOTE_A4, QUARTER_NOTE,
  NOTE_G4, HALF_NOTE,
  NOTE_REST, BREATH_MARK,
  //
  NOTE_C4, QUARTER_NOTE,
  NOTE_E4, QUARTER_NOTE,
  NOTE_G4, QUARTER_NOTE,
  NOTE_C5, QUARTER_NOTE,
  NOTE_E5, QUARTER_NOTE,
  NOTE_D5, QUARTER_NOTE,
  NOTE_E5, QUARTER_NOTE,
  NOTE_G5, QUARTER_NOTE,
  NOTE_E5, HALF_NOTE,
  NOTE_D5, HALF_NOTE,
  NOTE_C5, WHOLE_NOTE,
  NOTE_REST, BREATH_MARK };
// End BZ_SONG


#endif
```

6 ClockCode.ino

```
#include "TimerOne.h"
#include "timeClock.h"
#include "userInterface.h"
#include "buzzerDriver.h"
#include "globalParameters.h"

// Time marking variables
uint64_t tickCount = 0;
uint32_t lastTime = 0;
uint32_t elapsedTime = 0;
uint32_t timeMark = 0;

// Init function
void setup() {
  // Begin Serial
  Serial.begin(GP_BAUDRATE);
  // Init the functions
  BH_initAll();
  MX_init();
  BZ_init();
  // Starts the Interupts
  Timer1.initialize(GB_INTERUPT_PERIOD_US+GB_INTERUPT_TUNE_FACTOR);
  Timer1.attachInterrupt(mainISR);
}

// Prints the elapsed time (used in timing tuning)
void printElapsedTime()
{
  timeMark = micros();
  elapsedTime = timeMark-lastTime;
  lastTime = timeMark;
  Serial.println(elapsedTime);
}

// Prints the current time over serial
void printTime(timePiece* TmPc)
{
  char dispString[TC_TIME_LENGTH_STRING] = {0};
  timeClock_getTime(TmPc,dispString);
  for(uint8_t m = 0; m < TC_TIME_LENGTH_STRING; m++)
  {
    Serial.print(dispString[m]);
  }
  Serial.println("␣");
}

// Idle loop
void loop() {
}

// The main interupt routine
void mainISR()
{
  // Print the elapsed time over Serial
  //printElapsedTime();
  // Counter
  tickCount++;
  // Tick Clock if enabled
  if (ui_getTickStatus())
    timeClock_tickFWD(timeClock_getClock(),GB_INTERUPT_PERIOD_US/1000,1,1,1);
  // Tick the User Interface
  ui_tick();

  // Print the current time over Serial
  //printTime(timeClock_getClock());
}
```

7 globalParameters.h

```
#ifndef globalParameters_h
#define globalParameters_h

// Serial Rate on USB
#define GP_BAUDRATE 9600

// Nominal Interupt Period in micro seconds
#define GB_INTERUPT_PERIOD_US 50000

// Interupts per second
#define GB_INTERUPTS_PER_SECOND (1E6/GB_INTERUPT_PERIOD_US)

// Determined empiracally to tune the clock
/*
// +141 is slow
// +0 is fast#d
// +50 is fast
// +100 is fast
// +104 is fast
// +105 is fast 2 seconds over 15 hours
// +106 is slow
// +108 is slow
// +115 is slow
// +128 is slow
*/
#define GB_INTERUPT_TUNE_FACTOR (+106)



#endif
```

8 max7221Driver.cpp

```
#ifndef max7221Driver_c
#define max7221Driver_c

/*
  max7221Driver.h - Communicates with the MAX7721

  Created by Jakob Kunzler 07/04/2018
*/

#include "max7221Driver.h"

// Parameters
#define MX_DATA_LEN 16

// ADDRESS CODES //////////
///////////////////////////

// Defualt
#define MX_ADDR_NO_OPT 0x0

// Brightness
#define MX_ADDR_INTENSITY 0x0A
#define MX_MAX_BRIGHT 15
#define MX_MIN_BRIGHT 1

// Power
#define MX_ADDR_SHUTDOWN 0x0C
#define MX_DATA_OFF 0x00
#define MX_DATA_ON 0x01

// Display Test
#define MX_ADDR_DISPTEST 0x0F
#define MX_DATA_DISPTEST_NORMAL 0x00
#define MX_DATA_DISPTEST_MODE 0x01

// Code B decode
#define MX_ADDR_DECODE_MODE 0x09
#define MX_DATA_NO_DECODE 0x00
#define MX_DATA_CODEB_FLAG_DG0 0x01
#define MX_DATA_CODEB_FLAG_DG3_DG0 0x0F
#define MX_DATA_CODEB_FLAG_DG7_DG0 0xFF

#define MX_DATA_CODEB_CHAR_0 0x00
#define MX_DATA_CODEB_CHAR_1 0x01
#define MX_DATA_CODEB_CHAR_2 0x02
#define MX_DATA_CODEB_CHAR_3 0x03
#define MX_DATA_CODEB_CHAR_4 0x04
#define MX_DATA_CODEB_CHAR_5 0x05
#define MX_DATA_CODEB_CHAR_6 0x06
#define MX_DATA_CODEB_CHAR_7 0x07
#define MX_DATA_CODEB_CHAR_8 0x08
#define MX_DATA_CODEB_CHAR_9 0x09
#define MX_DATA_CODEB_CHAR_DASH 0x0A
#define MX_DATA_CODEB_CHAR_E 0x0B
#define MX_DATA_CODEB_CHAR_H 0x0C
```

```
#define MX_DATA_CODEB_CHAR_L 0x0D
#define MX_DATA_CODEB_CHAR_P 0x0E
#define MX_DATA_CODEB_CHAR_BLANK 0x0F

// Scan size
#define MX_ADDR_SCANLIM 0x0B

// Decimal Point
#define MX_MASK_DP 0x80

// Start Up
#define MX_STARTUP_LENGTH 8
const char MX_STARTUP[MX_STARTUP_LENGTH] = {'G','O','C','O','U','G','S','!'};

// Forms a data stream from an address and the data
uint16_t _MX_formCode(uint8_t address,uint8_t data)
{
  return (address<<8)|data;
}

// Returns a seven segment data code for a given char
uint8_t _MX_decodeChar(char d)
{

// Character Codes
#define MX_CHAR_0 0x7E
#define MX_CHAR_1 0x30
#define MX_CHAR_2 0x6D
#define MX_CHAR_3 0x79
#define MX_CHAR_4 0x33
#define MX_CHAR_5 0x5B
#define MX_CHAR_6 0x5F
#define MX_CHAR_7 0x70
#define MX_CHAR_8 0x7F
#define MX_CHAR_9 0x73
#define MX_CHAR_DASH 0x01
#define MX_CHAR_A 0x77
#define MX_CHAR_B 0x7F
#define MX_CHAR_C 0x4E
#define MX_CHAR_D 0x7E
#define MX_CHAR_E 0x4F
#define MX_CHAR_F 0x47
#define MX_CHAR_G 0x5F
#define MX_CHAR_H 0x37
#define MX_CHAR_I 0x06
#define MX_CHAR_J 0x3C
#define MX_CHAR_K 0x37
#define MX_CHAR_L 0x0E
#define MX_CHAR_M 0x54
#define MX_CHAR_N 0x76
#define MX_CHAR_O 0x7E
#define MX_CHAR_P 0x67
#define MX_CHAR_Q 0xFE
#define MX_CHAR_R 0x66
#define MX_CHAR_S 0x5B
#define MX_CHAR_T 0x07
#define MX_CHAR_U 0x3E
```

```
#define MX_CHAR_V 0X1C
#define MX_CHAR_W 0x2A
#define MX_CHAR_X 0x37
#define MX_CHAR_Y 0x3B
#define MX_CHAR_Z 0x6D
#define MX_CHAR_BLANK 0x00
#define MX_CHAR_EXCLAMATION_POINT 0xA0


  // Pick the character, and return the appropriate code
  switch(d){
    case '0':
    return MX_CHAR_0;
    break;

    case '1':
    return MX_CHAR_1;
    break;

    case '2':
    return MX_CHAR_2;
    break;

    case '3':
    return MX_CHAR_3;
    break;

    case '4':
    return MX_CHAR_4;
    break;

    case '5':
    return MX_CHAR_5;
    break;

    case '6':
    return MX_CHAR_6;
    break;

    case '7':
    return MX_CHAR_7;
    break;

    case '8':
    return MX_CHAR_8;
    break;

    case '9':
    return MX_CHAR_9;
    break;

    case '-':
    return MX_CHAR_DASH;
    break;

    case 'A':
    return MX_CHAR_A;
```

```
break;

case ’B’:
return MX_CHAR_B;
break;

case ’C’:
return MX_CHAR_C;
break;

case ’D’:
return MX_CHAR_D;
break;

case ’E’:
return MX_CHAR_E;
break;

case ’F’:
return MX_CHAR_F;
break;

case ’G’:
return MX_CHAR_G;
break;

case ’H’:
return MX_CHAR_H;
break;

case ’I’:
return MX_CHAR_I;
break;

case ’J’:
return MX_CHAR_J;
break;

case ’K’:
return MX_CHAR_K;
break;

case ’L’:
return MX_CHAR_L;
break;

case ’M’:
return MX_CHAR_M;
break;

case ’N’:
return MX_CHAR_N;
break;

case ’O’:
return MX_CHAR_O;
break;
```

```
    case ’P’:
    return MX_CHAR_P;
    break;

    case ’Q’:
    return MX_CHAR_Q;
    break;

    case ’R’:
    return MX_CHAR_R;
    break;

    case ’S’:
    return MX_CHAR_S;
    break;

    case ’T’:
    return MX_CHAR_T;
    break;

    case ’U’:
    return MX_CHAR_U;
    break;

    case ’V’:
    return MX_CHAR_V;
    break;

    case ’W’:
    return MX_CHAR_W;
    break;

    case ’X’:
    return MX_CHAR_X;
    break;

    case ’Y’:
    return MX_CHAR_Y;
    break;

    case ’Z’:
    return MX_CHAR_Z;
    break;

    case ’!’:
    return MX_CHAR_EXCLAMATION_POINT;
    break;

    default:
    return 0x00;
    break;
  }

}

// Adds the decimal place to a character code
```

```
uint16_t _MX_add_Decimal(uint16_t code)
{
  return (MX_MASK_DP|code);
}


/*
 Gets the bits in data between pos_start and pos_end and places at end.
 To get 1 bit, set pos_start = pos_end.
 */
uint8_t _MX_getBits(uint16_t data,uint8_t pos_end,int8_t pos_start)
{
  // Amount to shift left
  uint8_t left = MX_DATA_LEN-1-pos_end;
  // Amount to shift right
  uint8_t right = MX_DATA_LEN-1-(pos_end-pos_start);
  // shift to exclude unwanted data
  data = data<<left;
  data = data>>right;


  return data;
}


/*
  Sends an array uint8_t length of MX_DATA_LEN
  Each element is a 'bit' = to 1 or 0.
  D0 is the first element of the array
  Give the data in [D0,D1,...] format.
 */
void _MX_SendData(uint16_t data)
{
  digitalWrite(PIN_DATA_CLK,LOW);
  digitalWrite(PIN_CS,LOW);
  for (uint8_t m = 0; m < MX_DATA_LEN; m++)
  {
    uint8_t p = MX_DATA_LEN-1-m;
    uint8_t b = _MX_getBits(data,p,p);
    digitalWrite(PIN_DATA_IN,b);
    digitalWrite(PIN_DATA_CLK,HIGH);
    digitalWrite(PIN_DATA_CLK,LOW);
  }
  digitalWrite(PIN_CS,HIGH);
}



// Turns on = true, off = false
void MX_powerSwitch(bool state)
{
  if(state)
    _MX_SendData(_MX_formCode(MX_ADDR_SHUTDOWN,MX_DATA_ON));
  else
    _MX_SendData(_MX_formCode(MX_ADDR_SHUTDOWN,MX_DATA_OFF));
}
```

```
// Sets brightness between 0 and 15
void MX_setBrightness(uint8_t brightness)
{
  uint8_t level = _MX_getBits(brightness,3,0);
  _MX_SendData(_MX_formCode(MX_ADDR_INTENSITY,level));
}

// The number of segments to enable (0-7)
void MX_setNoSegments(uint8_t numSegs)
{
  _MX_SendData(_MX_formCode(MX_ADDR_SCANLIM,numSegs));
}

// The number of segments to enable (0-7)
void MX_noDecode()
{
  _MX_SendData(_MX_formCode(MX_ADDR_DECODE_MODE,MX_DATA_NO_DECODE));
}

// Toggle display test
void MX_dispTest(bool on)
{
  if(on)
  _MX_SendData(_MX_formCode(MX_ADDR_DISPTEST,MX_DATA_DISPTEST_MODE));
  else
  _MX_SendData(_MX_formCode(MX_ADDR_DISPTEST,MX_DATA_DISPTEST_NORMAL));
}

// Displays the chars in the string on the screen. Handles decimal point
void MX_disp_string(char* text,uint8_t textLength)
{
  uint8_t digitCtr = 0;
  for (uint8_t charCtr = 0; charCtr < textLength; charCtr++)
  {
    // Get Value
    char character = text[charCtr];
    // Get Code
    uint16_t code = _MX_decodeChar(text[charCtr]);
    // Peak ahead for decimal point
    if (charCtr+1 < textLength)
    {
      // Handle Decimal Point
      if (text[charCtr+1]==’.’)
      {
        // Add decimal
        code = _MX_add_Decimal(code);
        // Skip to next
        charCtr++;
      }
    }
    // Send
    _MX_SendData(_MX_formCode(digitCtr+1,code));
    digitCtr++;
  }
  while (digitCtr<8)
  {
    _MX_SendData(_MX_formCode(digitCtr+1,_MX_decodeChar(’␣’)));
    digitCtr++;
  }
}

// Send a blank screen
void MX_writeBLANK()
{
  char blank[8] = {'␣','␣','␣','␣','␣','␣','␣','␣',};
  MX_disp_string(blank,8);
}

// Init Settings
void MX_init()
{
  // INIT PINS
  pinMode(PIN_DATA_IN,OUTPUT);
  pinMode(PIN_DATA_CLK,OUTPUT);
  pinMode(PIN_CS,OUTPUT);

  // Initial Setting for screen
  MX_noDecode();
  MX_setNoSegments(MX_NUM_SEGMENTS-1);
  MX_setBrightness(MX_MIN_BRIGHT);
  MX_disp_string(MX_STARTUP,MX_STARTUP_LENGTH);
  MX_dispTest(false);
  MX_powerSwitch(true);
}

#endif
```

9 max7221Driver.h

```
#ifndef max7221Driver_h
#define max7221Driver_h

/*
  max7221Driver.h - Communicates with the MAX7721

  Created by Jakob Kunzler 07/04/2018
*/

#include "ArduinoStdInt.h"
#include "pinList.h"

// Parameters
#define MX_NUM_SEGMENTS 8 // Number of segments to use

// Turns on = true, off = false
void MX_powerSwitch(bool state);

// Sets brightness between 0 and 15
void MX_setBrightness(uint8_t brightness);

// The number of segments to enable (0-7)
void MX_setNoSegments(uint8_t numSegs);

// Do not try and decode the characters for "code B"
void MX_noDecode();

// Toggle display test
void MX_dispTest(bool on);

// Displays the chars in the string on the screen. Handles decimal point
void MX_disp_string(char* text,uint8_t textLength);

// Writes blank data to the screen
void MX_writeBLANK();

// Init Settings
void MX_init();

#endif
```

10 pinList.h

```
#ifndef pinList_h
#define pinList_h

// Buttons
#define PIN_BTN_0 A0
#define PIN_BTN_1 A1
#define PIN_BTN_2 A2
#define PIN_BTN_3 A3
#define PIN_BTN_4 A4
#define PIN_BTN_5 A5

// Switches
#define PIN_SW_0 A6

// Buzzer
#define PIN_BUZZER 6

// SPI to Seven Seg Controller
#define PIN_CS 5
#define PIN_DATA_CLK 4
#define PIN_DATA_IN 2

// UNASIGNED
#define PIN_EXTERN_INTERUPT 3
#define PIN_DIGITAL_7 7
#define PIN_DIGITAL_8 8
#define PIN_DIGITAL_9 9
#define PIN_DIGITAL_10 10
#define PIN_DIGITAL_11 11
#define PIN_DIGITAL_12 12
#define PIN_DIGITAL_13 13
#define PIN_ANALOG_7 A7

#endif
```

11 timeClock.cpp

```
/*
  timeClock.h - Keeps time as a state machine

  Created by Jakob Kunzler 07/04/2018
*/

#ifndef timeClock_c
#define timeClock_c

#include "timeClock.h"

// Constants for timekeeping
#define TC_SIXTYSECONDS 60
#define TC_SIXTYMINUTES 60
#define TC_TWENTYFOURHOURS 24
#define TC_TWELVEHOURS 12
#define TC_ZERO_UNDERFLOW 0
#define TC_ONETHOUSAND_MS 1000


// Two clocks
timePiece TIME_CLK;
timePiece ALARM_CLK;


/* Init the clock

  // tics per second: The number of times to call the tick function before one
      second passes
  // twelveHour_flag: 12 hour format = true, 24 hour format = false
  // seconds: Initial seconds, 0-59
  // minutes: Initial minutes, 0-59
  // hours: Initial hours in 24 hour format, 0-23
*/
void timeClock_init(timePiece* TmPc,int16_t ticksPerSec, bool twelveHour_flag,
    int8_t seconds, int8_t minutes, int8_t hours)
{
  TmPc->twelveHour_flag = twelveHour_flag;
  TmPc->milliSeconds = 0;
  TmPc->seconds = seconds;
  TmPc->minutes = minutes;
  TmPc->hours = hours;
}

// Moves clock forward the given amount
void timeClock_tickFWD(timePiece* TmPc,int16_t numMilSecs,int8_t numSecs,int8_t
    numMinutes,int8_t numHours)
{
  // Advance Milliseconds
  TmPc->milliSeconds = TmPc->milliSeconds + numMilSecs;
  if (TmPc->milliSeconds >= TC_ONETHOUSAND_MS)
  {
    // Roll Over
    TmPc->milliSeconds = TmPc->milliSeconds%TC_ONETHOUSAND_MS;
    // Advance Seconds
    TmPc->seconds = TmPc->seconds + numSecs;
    if (TmPc->seconds >= TC_SIXTYSECONDS)
    {
      // Roll Over
      TmPc->seconds = TmPc->seconds%TC_SIXTYSECONDS;
      // Advance Minutes
      TmPc->minutes = TmPc->minutes + numMinutes;
      if (TmPc->minutes >= TC_SIXTYMINUTES)
      {
        // Roll Over
        TmPc->minutes = TmPc->minutes%TC_SIXTYMINUTES;
        // Advance Hours
        TmPc->hours = TmPc->hours + numHours;
        if (TmPc->hours >= TC_TWENTYFOURHOURS)
        {
          // Roll Over
          TmPc->hours = TmPc->hours%TC_TWENTYFOURHOURS;
        }
      }
    }
  }
}

// Move the clock backward the given amount
void timeClock_tickREV(timePiece* TmPc,int16_t numMilSecs,int8_t numSecs,int8_t
    numMinutes,int8_t numHours)
{
  // Advance Milliseconds
  TmPc->milliSeconds = TmPc->milliSeconds - numMilSecs;
  if (TmPc->milliSeconds < TC_ZERO_UNDERFLOW)
  {
    // Roll Over
    TmPc->milliSeconds = (TC_ONETHOUSAND_MS-((-TmPc->milliSeconds)%
        TC_ONETHOUSAND_MS))%TC_ONETHOUSAND_MS;
    // Advance Seconds
    TmPc->seconds = TmPc->seconds - numSecs;
    if (TmPc->seconds < TC_ZERO_UNDERFLOW)
    {
      // Roll Over
      TmPc->seconds = (TC_SIXTYSECONDS-((-TmPc->seconds)%TC_SIXTYSECONDS))%
          TC_SIXTYSECONDS;
      // Advance Minutes
      TmPc->minutes = TmPc->minutes - numMinutes;
      if (TmPc->minutes < TC_ZERO_UNDERFLOW)
      {
        // Roll Over
        TmPc->minutes = (TC_SIXTYMINUTES-((-TmPc->minutes)%TC_SIXTYMINUTES))%
            TC_SIXTYMINUTES;
        // Advance Hours
        TmPc->hours = TmPc->hours - numHours;
        if (TmPc->hours < TC_ZERO_UNDERFLOW)
        {
          // Roll Over
          TmPc->hours = (TC_TWENTYFOURHOURS-((-TmPc->hours)%TC_TWENTYFOURHOURS))
              %TC_TWENTYFOURHOURS;
        }
      }
    }
  }
}

// Performs conversion on internal 24 hour, to external 12 hour format
uint8_t _timeClock_convert24hr_2_12hr(volatile uint8_t hour)
{
  if(hour == 0)
  {
    return 12;
  }
  if(hour > TC_TWELVEHOURS)
  {
    return hour-TC_TWELVEHOURS;
  }
  return hour;
}

// Decides AM or PM from the 24 hour internal system
char _timeClock_AM_or_PM(volatile uint8_t hour)
{
  if(hour == 0)
  {
    return ’A’;
  }
  if(hour >= TC_TWELVEHOURS)
  {
    return ’P’;
  }
  return ’A’;
}


// Update Current Time
// By doing this only when called, it save resources
void _timeClock_updateTime(timePiece* TmPc)
{
  if (TmPc->twelveHour_flag) // 12 Hours
  {
    sprintf(TmPc->currentTime, "%02u:%02u:%02u.%03u␣%cm",
        _timeClock_convert24hr_2_12hr(TmPc->hours), TmPc->minutes, TmPc->seconds,
         TmPc->milliSeconds, _timeClock_AM_or_PM(TmPc->hours));
  }
  else // 24 Hour
  {
    sprintf(TmPc->currentTime, "%02u:%02u:%02u.%03u␣␣␣", TmPc->hours, TmPc->
        minutes, TmPc->seconds, TmPc->milliSeconds);
  }
}

// Copies the current time into time
// time is an array of chars, TC_TIME_LENGTH_STRING long.
void _timeClock_AM_or_PM(timePiece* TmPc,char* timeString)
{
  // Update Time
  _timeClock_updateTime(TmPc);
  // Copy over
  for (uint8_t m = 0; m < TC_TIME_LENGTH_STRING; m++)
  {
    timeString[m] = TmPc->currentTime[m];
  }
}

// Copies the current time into time
// time is an array of chars, TC_TIME_LENGTH_STRING long.
void timeClock_getTime(timePiece* TmPc,char* timeString)
{
  // Update Time
  _timeClock_updateTime(TmPc);
  // Copy over
  for (uint8_t m = 0; m < TC_TIME_LENGTH_STRING; m++)
  {
    timeString[m] = TmPc->currentTime[m];
  }
}

// Get the alarm pointer
timePiece* timeClock_getAlarm()
{
  return &ALARM_CLK;
};

// Get the main clock pointer
timePiece* timeClock_getClock()
{
  return &TIME_CLK;
};

#endif
```

12 timeClock.h

```
/*
  timeClock.h - Keeps time as a state machine

  Created by Jakob Kunzler 07/04/2018
*/

#ifndef timeClock_h
#define timeClock_h

#include "ArduinoStdInt.h"
#include "Arduino.h"

// The length of the string containing current time
#define TC_TIME_LENGTH_STRING 16

// Data struct for the time clock
struct timePiece {
  int16_t milliSeconds;
  int8_t seconds; // Number of seconds
  int8_t minutes; // Number of minutes
  int8_t hours; // Number of hours (internally stored in 24 hour format)
  char currentTime[TC_TIME_LENGTH_STRING]; // String with the current time
  bool twelveHour_flag; // Twelve hour or no
};

// Get the alarm clock pointer
timePiece* timeClock_getAlarm();

// Get the time clock pointer
timePiece* timeClock_getClock();

/* Init the clock

  // tics per second: The number of times to call the tick function before one
      second passes
  // twelveHour_flag: 12 hour format = true, 24 hour format = false
  // seconds: Initial seconds, 0-59
  // minutes: Initial minutes, 0-59
  // hours: Initial hours in 24 hour format, 0-23
*/
void timeClock_init(timePiece* TmPc,int16_t ticksPerSec, bool twelveHour_flag,
    int8_t seconds, int8_t minutes, int8_t hours);

// Moves clock forward the given amount
void timeClock_tickFWD(timePiece* TmPc,int16_t numMilSecs,int8_t numSecs,int8_t
    numMinutes,int8_t numHours);

// Move the clock backward the given amount
void timeClock_tickREV(timePiece* TmPc,int16_t numMilSecs,int8_t numSecs,int8_t
    numMinutes,int8_t numHours);

// Copies the current time into time
// time is an array of chars, TC_TIME_LENGTH_STRING long.
void timeClock_getTime(timePiece* TmPc,char* timeString);
```

```
#endif
```

13 TimerOne.cpp

```
/*
 *  Interrupt and PWM utilities for 16 bit Timer1 on ATmega168/328
 *  Original code by Jesse Tane for http://labs.ideo.com August 2008
 *  Modified March 2009 by J r me Despatis and Jesse Tane for ATmega328
     support
 *  Modified June 2009 by Michael Polli and Jesse Tane to fix a bug in setPeriod
     () which caused the timer to stop
 *  Modified June 2011 by Lex Talionis to add a function to read the timer
 *  Modified Oct 2011 by Andrew Richards to avoid certain problems:
 *  - Add (long) assignments and casts to TimerOne::read() to ensure
     calculations involving tmp, ICR1 and TCNT1 aren't truncated
 *  - Ensure 16 bit registers accesses are atomic - run with interrupts disabled
      when accessing
 *  - Remove global enable of interrupts (sei())- could be running within an
     interrupt routine)
 *  - Disable interrupts whilst TCTN1 == 0.  Datasheet vague on this, but
     experiment shows that overflow interrupt
 *    flag gets set whilst TCNT1 == 0, resulting in a phantom interrupt.  Could
     just set to 1, but gets inaccurate
 *    at very short durations
 *  - startBottom() added to start counter at 0 and handle all interrupt
     enabling.
 *  - start() amended to enable interrupts
 *  - restart() amended to point at startBottom()
 * Modiied 7:26 PM Sunday, October 09, 2011 by Lex Talionis
 *  - renamed start() to resume() to reflect it's actual role
 *  - renamed startBottom() to start(). This breaks some old code that expects
     start to continue counting where it left off
 *
 *  This program is free software: you can redistribute it and/or modify
 *      it under the terms of the GNU General Public License as published by
 *      the Free Software Foundation, either version 3 of the License, or
 *      (at your option) any later version.
 *
 *      This program is distributed in the hope that it will be useful,
 *      but WITHOUT ANY WARRANTY; without even the implied warranty of
 *      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *      GNU General Public License for more details.
 *
 *      You should have received a copy of the GNU General Public License
 *      along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 *  See Google Code project http://code.google.com/p/arduino-timerone/ for
     latest
 */
#ifndef TIMERONE_cpp
#define TIMERONE_cpp

#include "TimerOne.h"

TimerOne Timer1;              // preinstatiate

ISR(TIMER1_OVF_vect)          // interrupt service routine that wraps a user
    defined function supplied by attachInterrupt
{
  Timer1.isrCallback();
}


void TimerOne::initialize(long microseconds)
{
  TCCR1A = 0;                 // clear control register A
  TCCR1B = _BV(WGM13);        // set mode 8: phase and frequency correct pwm,
      stop the timer
  setPeriod(microseconds);
}


void TimerOne::setPeriod(long microseconds)             // AR modified for
    atomic access
{

  long cycles = (F_CPU / 2000000) * microseconds;
                                     // the counter runs backwards after TOP,
      interrupt is at BOTTOM so divide microseconds by 2
  if(cycles < RESOLUTION)              clockSelectBits = _BV(CS10);
                   // no prescale, full xtal
  else if((cycles >>= 3) < RESOLUTION) clockSelectBits = _BV(CS11);
                   // prescale by /8
  else if((cycles >>= 3) < RESOLUTION) clockSelectBits = _BV(CS11) | _BV(CS10);
       // prescale by /64
  else if((cycles >>= 2) < RESOLUTION) clockSelectBits = _BV(CS12);
                   // prescale by /256
  else if((cycles >>= 2) < RESOLUTION) clockSelectBits = _BV(CS12) | _BV(CS10);
       // prescale by /1024
  else        cycles = RESOLUTION - 1, clockSelectBits = _BV(CS12) | _BV(CS10);
       // request was out of bounds, set as maximum

  oldSREG = SREG;
  cli();                                                        // Disable
      interrupts for 16 bit register access
  ICR1 = pwmPeriod = cycles;                                          // ICR1 is
       TOP in p & f correct pwm mode
  SREG = oldSREG;

  TCCR1B &= ~(_BV(CS10) | _BV(CS11) | _BV(CS12));
  TCCR1B |= clockSelectBits;                                          // reset
      clock select register, and starts the clock
}

void TimerOne::setPwmDuty(char pin, int duty)
{
  unsigned long dutyCycle = pwmPeriod;

  dutyCycle *= duty;
  dutyCycle >>= 10;

  oldSREG = SREG;
  cli();
  if(pin == 1 || pin == 9)       OCR1A = dutyCycle;
  else if(pin == 2 || pin == 10) OCR1B = dutyCycle;
  SREG = oldSREG;
```

```
}

void TimerOne::pwm(char pin, int duty, long microseconds)  // expects duty cycle
     to be 10 bit (1024)
{
  if(microseconds > 0) setPeriod(microseconds);
  if(pin == 1 || pin == 9) {
    DDRB |= _BV(PORTB1);                                   // sets data
        direction register for pwm output pin
    TCCR1A |= _BV(COM1A1);                                 // activates the
        output pin
  }
  else if(pin == 2 || pin == 10) {
    DDRB |= _BV(PORTB2);
    TCCR1A |= _BV(COM1B1);
  }
  setPwmDuty(pin, duty);
  resume();                     // Lex - make sure the clock is running.  We don
      't want to restart the count, in case we are starting the second WGM
                                        // and the first one is in the middle of
                                             a cycle
}

void TimerOne::disablePwm(char pin)
{
  if(pin == 1 || pin == 9)       TCCR1A &= ~_BV(COM1A1);   // clear the bit that
       enables pwm on PB1
  else if(pin == 2 || pin == 10) TCCR1A &= ~_BV(COM1B1);   // clear the bit that
       enables pwm on PB2
}

void TimerOne::attachInterrupt(void (*isr)(), long microseconds)
{
  if(microseconds > 0) setPeriod(microseconds);
  isrCallback = isr;                                       // register the user'
      s callback with the real ISR
  TIMSK1 = _BV(TOIE1);                                     // sets the timer
      overflow interrupt enable bit
        // might be running with interrupts disabled (eg inside an ISR), so don'
            t touch the global state
//  sei();
  resume();
}

void TimerOne::detachInterrupt()
{
  TIMSK1 &= ~_BV(TOIE1);                                   // clears the timer
      overflow interrupt enable bit
```

```
}

void TimerOne::resume()                         // AR suggested
{
  TCCR1B |= clockSelectBits;
}

void TimerOne::restart()                // Depricated - Public interface to
    start at zero - Lex 10/9/2011
{
        start();
}

void TimerOne::start()  // AR addition, renamed by Lex to reflect it's actual
    role
{
  unsigned int tcnt1;

  TIMSK1 &= ~_BV(TOIE1);        // AR added
  GTCCR |= _BV(PSRSYNC);                // AR added - reset prescaler (NB:
      shared with all 16 bit timers);

  oldSREG = SREG;                               // AR - save status register
  cli();                                                // AR - Disable
      interrupts
  TCNT1 = 0;
  SREG = oldSREG;                       // AR - Restore status register
        resume();
  do {  // Nothing -- wait until timer moved on from zero - otherwise get a
      phantom interrupt
        oldSREG = SREG;
        cli();
        tcnt1 = TCNT1;
        SREG = oldSREG;
  } while (tcnt1==0);

//  TIFR1 = 0xff;                       // AR - Clear interrupt flags
//  TIMSK1 = _BV(TOIE1);              // sets the timer overflow interrupt
    enable bit
}

void TimerOne::stop()
{
  TCCR1B &= ~(_BV(CS10) | _BV(CS11) | _BV(CS12));          // clears all clock
      selects bits
}

unsigned long TimerOne::read()          //returns the value of the timer in
```

```
    microseconds
{                                                                        //rember
    ! phase and freq correct mode counts up to then down again
        unsigned long tmp;                               // AR amended to hold
            more than 65536 (could be nearly double this)
        unsigned int tcnt1;                              // AR added

        oldSREG= SREG;
        cli();
        tmp=TCNT1;
        SREG = oldSREG;

        char scale=0;
        switch (clockSelectBits)
        {
        case 1:// no prescalse
                scale=0;
                break;
        case 2:// x8 prescale
                scale=3;
                break;
        case 3:// x64
                scale=6;
                break;
        case 4:// x256
                scale=8;
                break;
        case 5:// x1024
                scale=10;
                break;
        }

        do {    // Nothing -- max delay here is ~1023 cycles.  AR modified
                oldSREG = SREG;
                cli();
                tcnt1 = TCNT1;
                SREG = oldSREG;
        } while (tcnt1==tmp); //if the timer has not ticked yet

        //if we are counting down add the top value to how far we have counted
            down
        tmp = (  (tcnt1>tmp) ? (tmp) : (long)(ICR1-tcnt1)+(long)ICR1  );
                    // AR amended to add casts and reuse previous TCNT1
        return ((tmp*1000L)/(F_CPU /1000L))<<scale;
}

#endif
```

14 TimerOne.h

```
/*
 *  Interrupt and PWM utilities for 16 bit Timer1 on ATmega168/328
 *  Original code by Jesse Tane for http://labs.ideo.com August 2008
 *  Modified March 2009 by J r me Despatis and Jesse Tane for ATmega328
     support
 *  Modified June 2009 by Michael Polli and Jesse Tane to fix a bug in setPeriod
     () which caused the timer to stop
 *  Modified June 2011 by Lex Talionis to add a function to read the timer
 *  Modified Oct 2011 by Andrew Richards to avoid certain problems:
 *  - Add (long) assignments and casts to TimerOne::read() to ensure
     calculations involving tmp, ICR1 and TCNT1 aren't truncated
 *  - Ensure 16 bit registers accesses are atomic - run with interrupts disabled
      when accessing
 *  - Remove global enable of interrupts (sei())- could be running within an
     interrupt routine)
 *  - Disable interrupts whilst TCTN1 == 0.  Datasheet vague on this, but
     experiment shows that overflow interrupt
 *    flag gets set whilst TCNT1 == 0, resulting in a phantom interrupt.  Could
     just set to 1, but gets inaccurate
 *    at very short durations
 *  - startBottom() added to start counter at 0 and handle all interrupt
     enabling.
 *  - start() amended to enable interrupts
 *  - restart() amended to point at startBottom()
 * Modiied 7:26 PM Sunday, October 09, 2011 by Lex Talionis
 *  - renamed start() to resume() to reflect it's actual role
 *  - renamed startBottom() to start(). This breaks some old code that expects
     start to continue counting where it left off
 *
 *  This program is free software: you can redistribute it and/or modify
 *      it under the terms of the GNU General Public License as published by
 *      the Free Software Foundation, either version 3 of the License, or
 *      (at your option) any later version.
 *
 *      This program is distributed in the hope that it will be useful,
 *      but WITHOUT ANY WARRANTY; without even the implied warranty of
 *      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *      GNU General Public License for more details.
 *
 *      You should have received a copy of the GNU General Public License
 *      along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 *  See Google Code project http://code.google.com/p/arduino-timerone/ for
     latest
 */
#ifndef TIMERONE_h
#define TIMERONE_h

#include <avr/io.h>
#include <avr/interrupt.h>

#define RESOLUTION 65536    // Timer1 is 16 bit

class TimerOne
{
  public:

    // properties
    unsigned int pwmPeriod;
    unsigned char clockSelectBits;
        char oldSREG;                                   // To hold Status
            Register while ints disabled

    // methods
    void initialize(long microseconds=1000000);
    void start();
    void stop();
    void restart();
        void resume();
        unsigned long read();
    void pwm(char pin , int duty , long microseconds=-1);
    void disablePwm(char pin);
    void attachInterrupt(void (*isr)() , long microseconds=-1);
    void detachInterrupt();
    void setPeriod(long microseconds);
    void setPwmDuty(char pin , int duty);
    void (*isrCallback)();
};

extern TimerOne Timer1;
#endif
```

15 userInterface.cpp

```
#ifndef userInterface_c
#define userInterface_c

// Includes
#include "userInterface.h"
#include "timeClock.h"
#include "max7221Driver.h"

// General Parameters
#define DISP_STRING_LEN 10

// States
#define UI_INIT_S 0
#define UI_IDLE_S 1
#define UI_DRIVE_ALARM_S 2
#define UI_SET_TIME_SECONDS_S 3
#define UI_SET_TIME_MINUTES_S 4
#define UI_SET_TIME_HOURS_S 5
#define UI_SET_TIME_AMPM_S 6
#define UI_SET_BUTTON_RELEASE_S 7
#define UI_SET_FLASH_S 8

// Button Aliases
#define UI_B0 (BH_getAnIO(’B’,0))
#define UI_B1 (BH_getAnIO(’B’,1))
#define UI_B2 (BH_getAnIO(’B’,2))
#define UI_B3 (BH_getAnIO(’B’,3))
#define UI_B4 (BH_getAnIO(’B’,4))
#define UI_B5 (BH_getAnIO(’B’,5))
#define UI_SW0 (BH_getAnIO(’S’,0))

// Counters
uint16_t ui_updateDisp_ctr = 0;
uint16_t ui_flashDisp_ctr = 0;

// State Variables
uint8_t ui_currentState = UI_INIT_S;
uint8_t ui_flashLastState = UI_SET_TIME_SECONDS_S;
uint8_t ui_buttonRelease_NextState = UI_INIT_S;
uint8_t ui_buttonRelease_ButtonNum = 0;

// The time piece to set
timePiece* settingTimePiece = timeClock_getClock();

// Status Flags
bool soundAlarm_flag = false;
bool tickClock_flag = true;

// Time on display as a string
char timeOnDisplay[DISP_STRING_LEN] = {0};
///////////////////////////////////////////////


// Saves the display time as timeOnDisplay
void _ui_storeDispTime(char* disp_time)
```

```
{
  for (uint8_t m = 0; m < DISP_STRING_LEN; m++)
    timeOnDisplay[m] = disp_time[m];
}


// A debug print
void _ui_printDisp(char* dispString)
{
  for (uint8_t m = 0; m < DISP_STRING_LEN; m++)
  {
    Serial.print(dispString[m]);
  }
}

// Updates the display
void ui_updateDisplay(timePiece* TmPc)
{
  // Get the current time
  char timeString[TC_TIME_LENGTH_STRING] = {0};
  timeClock_getTime(TmPc,timeString);

  // Extract characters
  char dispString[DISP_STRING_LEN] = {0};
  dispString[0] = timeString[0];
  dispString[1] = timeString[1];
  dispString[2] = ’.’;
  dispString[3] = timeString[3];
  dispString[4] = timeString[4];
  dispString[5] = ’.’;
  dispString[6] = timeString[6];
  dispString[7] = timeString[7];
  dispString[8] = ’␣’;

  // Check Format from switch
  if (TmPc->twelveHour_flag) // High =  12 hour
    dispString[9] = timeString[13];
  else // Low = 24 hour
    dispString[9] = ’␣’;


  // Print Text to Seven Segment
  MX_disp_string(dispString, DISP_STRING_LEN);
  // Store Time
  _ui_storeDispTime(dispString);
}



// Checks if the alarm should go
bool _ui_checkForAlarmTrigger()
{
  // Check if alarm is set
  if (BH_getAnIO(’S’, 0) == 0)
    return false;

  // Compare the TM and AL object
  if(timeClock_getClock()->hours != timeClock_getAlarm()->hours)
    return false;
  if(timeClock_getClock()->minutes != timeClock_getAlarm()->minutes)
    return false;
  if(timeClock_getClock()->seconds != timeClock_getAlarm()->seconds)
    return false;
  // Times match (not milliseconds)
  return true;
}


// Tick function for the user interface
void ui_tick()
{
  // State Machine
  switch (ui_currentState)
  {

    //////////////////////
    // Intialize everything
    case (UI_INIT_S):
      // Action //
      // Start clock
      timeClock_init(timeClock_getClock(),GB_INTERUPTS_PER_SECOND, UI_12HR_FLAG,
           UI_CLK_START_SECONDS, UI_CLK_START_MINUTES, UI_CLK_START_HOURS);
      // Start alarm
      timeClock_init(timeClock_getAlarm(),GB_INTERUPTS_PER_SECOND, UI_12HR_FLAG,
           UI_ALARM_START_SECONDS, UI_ALARM_START_MINUTES, UI_ALARM_START_HOURS);
      ui_updateDisp_ctr = 0;
      tickClock_flag = true;
      // Pre-seed at half duty.
      ui_flashDisp_ctr = (GB_INTERUPTS_PER_SECOND/2);
      // Advance //
      ui_currentState = UI_IDLE_S;
      break;

    ///////////////
    // Waiting state
    case (UI_IDLE_S):
      // Action //
      ui_updateDisp_ctr++;

      // Advance //

      // Main Time Set Mode
      if (UI_B5)
      {
        ui_currentState = UI_SET_BUTTON_RELEASE_S;
        // Set up the button release parameters
        ui_buttonRelease_ButtonNum = 5;
        // Do hours first
        ui_buttonRelease_NextState = UI_SET_TIME_HOURS_S;
        // Stop ticking clock
        tickClock_flag = false;
        // Setting the main time piece (TM)
        settingTimePiece = timeClock_getClock();
        break;
      }

      // Alarm Time Set Mode
      if (UI_B4)
      {
        ui_currentState = UI_SET_BUTTON_RELEASE_S;
        // Set up the button release parameters
        ui_buttonRelease_ButtonNum = 4;
        // Do hours first
        ui_buttonRelease_NextState = UI_SET_TIME_HOURS_S;
        // Keep ticking clock
        tickClock_flag = true;
        // Setting the main time piece (TM)
        settingTimePiece = timeClock_getAlarm();
        break;
      }

      // Check for alarm
      if (_ui_checkForAlarmTrigger())
      {
        ui_currentState = UI_DRIVE_ALARM_S;
        soundAlarm_flag = true;
        break;
      }

      // Update Display after so many ticks
      if (ui_updateDisp_ctr >= DISP_UPDATE_TICKS)
      {
        ui_currentState = UI_IDLE_S;
        ui_updateDisplay(timeClock_getClock());
        ui_updateDisp_ctr = 0;
        break;
      }
      break;

    ////////////////
    // Drive Alarm state
    case (UI_DRIVE_ALARM_S):
      // Action //

      // Advance //
      // Check if alarm switch is on
      if (BH_getAnIO(’S’, 0))
      {
        ui_currentState = UI_DRIVE_ALARM_S;
        soundAlarm_flag = true;
      }
      else // User turned off
      {
        ui_currentState = UI_IDLE_S;
        soundAlarm_flag = false;
      }
      // Update Alarm
      BZ_alarmBZ_SONG(soundAlarm_flag);
      break;
```

```
//////////////////////
// Wait for release
case (UI_SET_BUTTON_RELEASE_S):
  // If held, stay
  if (BH_getAnIO(’B’,ui_buttonRelease_ButtonNum))
  {
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    MX_dispTest(true);
  }
  else // Button released
  {
    ui_currentState = ui_buttonRelease_NextState;
    MX_dispTest(false);
  }
  break;

//////////////////////
// Set seconds
case (UI_SET_TIME_SECONDS_S):
  // Action //

  // Increase
  if (UI_B2)
  {
    timeClock_tickFWD(settingTimePiece,1000,1,0,0);
  }
  // Decrease
  else if (UI_B1)
  {
    timeClock_tickREV(settingTimePiece,1000,1,0,0);
  }


  // Advance //

  // Exit Set Mode Time Mode
  if ((UI_B5)&&(settingTimePiece==timeClock_getClock()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 5;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }
  // Exit Set Mode Alarm Mode
  if ((UI_B4)&&(settingTimePiece==timeClock_getAlarm()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 4;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }
  // Left
  if (UI_B3)
  {
    ui_buttonRelease_ButtonNum = 3;
    ui_buttonRelease_NextState = UI_SET_TIME_MINUTES_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }
  // Right
  if (UI_B0)
  {
    ui_buttonRelease_ButtonNum = 0;
    ui_buttonRelease_NextState = UI_SET_TIME_AMPM_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }


  // Flash if nothing else
  ui_flashLastState = UI_SET_TIME_SECONDS_S;
  ui_currentState = UI_SET_FLASH_S;
  break;

//////////////////////
// Set minutes
case (UI_SET_TIME_MINUTES_S):
  // Action //

  // Increase
  if (UI_B2)
  {
    timeClock_tickFWD(settingTimePiece,1000,60,1,0);
  }
  // Decrease
  else if (UI_B1)
  {
    timeClock_tickREV(settingTimePiece,1000,60,1,0);
  }

  // Advance //

  // Exit Set Mode Time Mode
  if ((UI_B5)&&(settingTimePiece==timeClock_getClock()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 5;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }
  // Exit Set Mode Alarm Mode
  if ((UI_B4)&&(settingTimePiece==timeClock_getAlarm()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 4;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }

  // Left
  if (UI_B3)
  {
    ui_buttonRelease_ButtonNum = 3;
    ui_buttonRelease_NextState = UI_SET_TIME_HOURS_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }
  // Right
  if (UI_B0)
  {
    ui_buttonRelease_ButtonNum = 0;
    ui_buttonRelease_NextState = UI_SET_TIME_SECONDS_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }


  // Flash if nothing else
  ui_flashLastState = UI_SET_TIME_MINUTES_S;
  ui_currentState = UI_SET_FLASH_S;
  break;


//////////////////////
// Set Hours
case (UI_SET_TIME_HOURS_S):
  // Action //

  // Increase
  if (UI_B2)
  {
    timeClock_tickFWD(settingTimePiece,1000,60,60,1);
  }
  // Decrease
  else if (UI_B1)
  {
    timeClock_tickREV(settingTimePiece,1000,60,60,1);
  }

  // Advance //
  // Exit Set Mode Time Mode
  if ((UI_B5)&&(settingTimePiece==timeClock_getClock()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 5;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }
  // Exit Set Mode Alarm Mode
  if ((UI_B4)&&(settingTimePiece==timeClock_getAlarm()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 4;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }

  // Left
  if (UI_B3)
  {
    ui_buttonRelease_ButtonNum = 3;
    ui_buttonRelease_NextState = UI_SET_TIME_AMPM_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }
  // Right
  if (UI_B0)
  {
    ui_buttonRelease_ButtonNum = 0;
    ui_buttonRelease_NextState = UI_SET_TIME_MINUTES_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }


  // Flash if nothing else
  ui_flashLastState = UI_SET_TIME_HOURS_S;
  ui_currentState = UI_SET_FLASH_S;
  break;

//////////////////////
// Set AM or PM
case (UI_SET_TIME_AMPM_S):
  // Action //

  // Increase
  if (UI_B2)
  {
    timeClock_tickFWD(settingTimePiece,1000,60,60,12);
  }
  // Decrease
  else if (UI_B1)
  {
    timeClock_tickREV(settingTimePiece,1000,60,60,12);
  }

  // Advance //

  // Exit Set Mode Time Mode
  if ((UI_B5)&&(settingTimePiece==timeClock_getClock()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 5;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }
  // Exit Set Mode Alarm Mode
  if ((UI_B4)&&(settingTimePiece==timeClock_getAlarm()))
  {
    // Set up the button release parameters
    ui_buttonRelease_ButtonNum = 4;
    ui_buttonRelease_NextState = UI_IDLE_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    // Start ticking clock
    tickClock_flag = true;
    break;
  }

  // Left
  if (UI_B3)
  {
    ui_buttonRelease_ButtonNum = 3;
    ui_buttonRelease_NextState = UI_SET_TIME_SECONDS_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }
  // Right
  if (UI_B0)
  {
    ui_buttonRelease_ButtonNum = 0;
    ui_buttonRelease_NextState = UI_SET_TIME_HOURS_S;
    ui_currentState = UI_SET_BUTTON_RELEASE_S;
    break;
  }

  // Flash if nothing else
  ui_flashLastState = UI_SET_TIME_AMPM_S;
  ui_currentState = UI_SET_FLASH_S;
  break;


//////////////////////
// MAKE FLASHING
case (UI_SET_FLASH_S):
  // Action //
  ui_flashDisp_ctr= (ui_flashDisp_ctr+1)%int(GB_INTERUPTS_PER_SECOND);
  if(ui_flashDisp_ctr<DISP_FLASH_TICKS)
    MX_writeBLANK();
  else
    ui_updateDisplay(settingTimePiece);
  // Advance //
  ui_currentState = ui_flashLastState;
  break;
```

```
    ///////
    default:
      ui_currentState = UI_INIT_S;
      break;
  }
}

// Get the alarm status
bool ui_getAlarmStatus()
{
  return soundAlarm_flag;
}

// Get the ticking status
bool ui_getTickStatus()
{
  return tickClock_flag;
}


#endif
```

16 userInterface.h

```
#ifndef userInterface_h
#define userInterface_h

// Includes
#include "buttonHandler.h"
#include "timeClock.h"
#include "max7221Driver.h"
#include "buttonHandler.h"
#include "buzzerDriver.h"
#include "globalParameters.h"

// Clock Start Parameters
#define UI_CLK_START_HOURS 0
#define UI_CLK_START_MINUTES 0
#define UI_CLK_START_SECONDS 0

// Twelve Hour Format, or 24 hour?
#define UI_12HR_FLAG (true)

// Alarm Start Parameters
#define UI_ALARM_START_HOURS 0
#define UI_ALARM_START_MINUTES 0
#define UI_ALARM_START_SECONDS 5

// State Timing
#define DISP_UPDATE_TICKS (GB_INTERUPTS_PER_SECOND)
#define DISP_FLASH_TICKS (GB_INTERUPTS_PER_SECOND*1/20)

// Updates the display with data from the time piece
void ui_updateDisplay(timePiece* TmPc);

// Tick function for the user interface
void ui_tick();

// Get the alarm status
bool ui_getAlarmStatus();

// Get the ticking status
bool ui_getTickStatus();



#endif
```