H04L2A
Software for Real-Time and Embedded Systems

Project 1

Alarm Clock on a PIC18F97J60-based Devboard

Group 14
Olle Calderon
Orsolya Lukács-Kisbandi

1 Introduction

An alarm clock was implemented on a development board based on the PIC18F97J60 microprocessor by Microchip. The alarm clock had to have the following features:

1. The format of the time displayed had to be hh:mm:ss, which meant that the display had to be updated once a second
2. Hours had to be counted from 0 to 23; so the display had to jump from 23:59.59 to 00:00.00
3. The clock time and alarm time had to be set when the board was powered up. Ringing had to be replaced by blinking an LED every second for 30 seconds. To achieve this, a timer had to be configured on the PIC to generate interrupts, making it possible to turn on/off an LED twice a second and update a seconds counter once every second
4. While the clock was running, it had to be possible to change the alarm time and the the current time without influencing the clock

2 Documentation

The aim of the project was to design the alarm clock in the C programming language, running on the microprocessor without an operating system. On this microprocessor, the program had direct access to the memory and the peripherals.

2.1 Timer configuration

The PIC processor featured two timer modules; Timer0 and Timer1. In this project, Timer0 was used. In order to generate interrupts on timer0 overflow, the interrupt registers of the PIC were configured accordingly:

```
INTCONbits.GIE = 1;   // Enable global interrupts
INTCONbits.TMR0IE=0;  // Enable timer0 interrupts
INTCON2bits.TMR0IP=1; // TMR0 has high priority
```

By setting/resetting TMROIE in the INTCON register, it was possible to make the time start/stop on the clock. This feature was useful e.g. on startup, before the time was set by the user. Timer0 is a 16bit timer that increases its value for every instruction clock cycle (one fourth of the system clock f_{CLK}). When the timer overflows, an interrupt is generated. In order to get precisely one (and one half second, since the LED had to change its value every half second), the timer's value had to be set accordingly. The number of iterations of the instruction clock to measure half a second was

$$n_{TMR0} = \frac{f_{CLK}}{2 \cdot 4} = 3125000$$

This value was larger than 2^{16}, which made it impossible to implement solely using a 16bit timer. To solve this problem without a prescaler, a value had to be loaded into the timer to generate an interrupt at an integer fraction of half a second. It was found that a timer value of 25000 and a factor of 125 would equal exactly n_{TMR0}. Thus, with the timer set to overflow and generate an interrupt every 25000th instruction cycle, every 125th interrupt would equal to a half second.

```
#define DIVISOR         25000
#define HIGH            ((0xFFFF - DIVISOR) & 0xFF00) >> 8
#define LOW             (0xFFFF - DIVISOR) & 0xFF
```

The high and low bytes (HIGH and LOW above) were loaded into the timer. The TOCON (Timer0 control) register was configured to accomodate the desired functionality.

```
TMROH = HIGH;           // Load timer high byte
TMROL = LOW;            // Load timer low byte
TOCONbits.TO8BIT = 0;   // 16bit
TOCONbits.TOCS = 0;     // Clock source = instruction cycle CLK
TOCONbits.TOSE = 0;     // Rising edge
TOCONbits.PSA = 1;      // No prescaler
TOCONbits.TMROON = 1;   // Start timer
```

2.2 Interrupt routine

The basic functionality of the interrupt routine to enable increasing the second-counter was implemented as follows:

```
void high_isr (void) interrupt 1
{
  static unsigned char ticks = 0;
  if(INTCONbits.TMROIF)          // If TMRO flag is set
  {
    INTCONbits.TMROIE = 0;       // Disable TMRO interrupts
    if (++ticks == 250)
    {
      /* Increase second-counter */
      ticks = 0;
      /* If set alarm time is same as current time, trigger alarm */
    }
    INTCONbits.TMROIF = 0;       // Reset TMRO flag
    INTCONbits.TMROIE = 1;       // Re-enable TMRO interrupts
    TMROH=HIGH;                  // Reset TMRO values
    TMROL=LOW;
  }
}
```

Of course, the code above is just a rough example of how the ISR works. For the full code, see section 5.

2.3 Current and alarm time

A struct `Time` was created, consisting of three attributes; hours, minutes and seconds. Two instances of this struct were used; `current_time` containing the current time and `alarm_time` containing the alarm time.

2.4 Other implementation decisions

The functionality of the clock was implemented using a state machine. The implemented state machine was a simple one, with internal conditionals in each state. This was a design choice to keep the code easy to read and understand.
A global variable `hms` (short for **h**ours, **m**inutes, **s**econds) was used to keep track of which attribute of time was currently being changed. If `hms = 1`, hours could be changed, if `hms = 2` minutes and if `hms = 3` seconds. If `hms` $>$ `3`, the time configuration was done.
Also, the state machine kept track of which `mode` the alarm clock was currently in; `SET` (setting the current time), `ALARM` (setting alarm time) or `CURRENT` (displaying the current time).

These were the states present in the state machine:

- `STARTUP`
  Run at startup. Sets `current_time`, `alarm_time` and `hms` to zero, then goes to state `SET_TIME`.
- `SET_TIME`
  Disables timer interrupts when setting current time.
  Invokes function `change_time()`, which sets `current_time` and/or `alarm_time` to the required values, depending on `hms` and what `mode` the clock is currently in. When both buttons are pressed, the next state is `WAIT_FOR_RELEASE`.
- `WAIT_FOR_RELEASE`
  When buttons are released, the next state to be reached is decided. If `hms` $\leq$ `3`, the time is set (if `mode = SET`, the current time is set and if `mode = ALARM`, the alarm time is set). If `hms` $>$ `3`, the time setup is done and the state `INC_TIME` is reached. The interrupts of Timer0 are also enabled at this point. However, on first startup, the alarm clock lets the user input first the current time and then the alarm time before going to `INC_TIME` and enabling interrupts.
- `INC_TIME`
  In this state, the time is running and the state machine is waiting for an external event (e.g. reaching alarm time). This state also makes sure that the variables of `current_time` are adhering to the rules of a 24-hour clock (i.e. that minutes are updated every 60 seconds, hours updated every 60 minutes, time is reset after 23:59.59 etc.).
  If both buttons are pressed in this state, the state machine goes to `CHOICE`, unless the alarm is triggered (i.e. the alarm time is reached).
- `CHOICE`
  Decides if `current_time` or `alarm_time` is to be set, based on user input. Sets `mode` accordingly, then goes to `SET_TIME`.

3 User manual

1. Inital setup

   (a) Set current time

   - First, set the `hours` by pressing and holding `BUT0` to decrement or `BUT1` to increment.
   - When done setting the `hours`, press both buttons to switch to `minutes`. Use the buttons to set the minutes just like the hours. Repeat the same procedure for `seconds`.
   - After setting `seconds`, press both buttons again to start setting the alarm time.

   (b) Set alarm time

   - Follow the same instructions as ***Set current time***. When seconds are set, press both buttons again. This will exit the time setup and just display the current time.

2. Changing current time after initial setup

   - Press both buttons at the same time.
   - Press `BUT0` (This will enter the set current time mode).
   - Set the time (as explained in ***Set current time***). The procedure is the same as explained in ***Set current time*** above, but after setting seconds and pressing both buttons, the clock goes back to just displaying the current time).

3. Changing alarm time after initial setup

   - Press both buttons at the same time.
   - Press `BUT1` (This will enter the set alarm time mode).
   - Set the alarm time (as explained in ***Set alarm time***). The procedure is the same as explained in ***Set alarm time*** above, but after setting seconds and pressing both buttons, the clock goes back to just displaying the current time).

4. Alarm

   - When the alarm time is reached, the red LED on the board is blinked once every second for 30 seconds. During this time, it is not possible to set the current or alarm time.
   - After 30 seconds and the red LED stops blinking, the current time will be displayed, regardless what the user was doing when the alarm was triggered. The alarm time previously entered will remain unchanged, which allows the user to be awaken at the time time every day.

4 Instructions

To compile the code, you need to install a few things on your computer. This guide is for UNIX and UNIX-like systems and these commands were succesfully used under Mac OSX[1] during the development of this code.

4.1 Prerequisites

First, you'll need to configure, make and install GPUTILS[2] and SDCC[3]. To do this, you might have to install `bison` and `flex` on your system (if the printouts in your terminal tells you these components are missing).

Now, go to `/usr/local/share/sdcc/lib/pic16` using your terminal and run these linker scripts:

```
ln -s libm18f.a libm18f.lib
ln -s libsdcc.a libsdcc.lib
ln -s libio18f97j60.a libio18f97j60.lib
ln -s libdev18f97j60.a libdev18f97j60.lib
```

After running the linker scripts, your system is ready to compile the program.

4.2 Compiling

If you've followed the previous steps, your system should now be able to compile using the included `Makefile`. Using your terminal, go to the directory containing `clock.c` and type the command `make clock`. If the compilation was successful, the file `clock.hex` (in the same directory as `clock.c`) should now be generated (or if it already existed, replaced by the new code). If this file is not generated, you might want to try entering `sdcc -mpic16 -p18f97j60 clock.c` followed by `sdcc -mpic16 -p18f97j60 -L/usr/local/lib/pic16 clock.c` instead.

4.3 Download program to board

If you're not already there, go to the folder containing `clock.hex` using your terminal. When the program is compiled, connect the devboard to your computer[4] using an ethernet cable. Run `tftp` in the terminal using IP address `192.168.97.60`, which is the IP address of the devboard. Enter `binary`, `trace` and `verbose` in the terminal when connected to the board using `tftp`. Type `put clock.hex` into the terminal, but don't hit enter just yet. Push the reset button on the board, wait one or two seconds for the board to connect to your network and then hit enter. If the program is successfully transferred to the board, you should see in the terminal that packets have been sent and that ACKs are received properly. At this point, the board will reset and run the program you just transferred.

[1] We used version 10.11.6, but other version should probably work the same.

[2] We used v0.13.7, but other versions might also work (not tested, your mileage may vary).

[3] We used SRC-20091215-5595, but other versions might also work (not tested).

[4] We used a router in between, but you may be able to connect directly. Regardless, you need to configure your network to allow connection between your PC and the devboard.

5 CODE

```c
#define __18F97J60
#define __SDCC__
#define THIS_INCLUDES_THE_MAIN_FUNCTION
#include "Include/HardwareProfile.h"

#include <string.h>
#include <stdlib.h>

#include "Include/LCDBlocking.h"
#include "Include/TCPIP_Stack/Delay.h"

#define DIVISOR 25000
#define HIGH ((0xFFFF - DIVISOR) & 0xFF00) >> 8
#define LOW (0xFFFF - DIVISOR) & 0xFF

const char *state2str[] =   //Used for debug
{
  "STARTUP",
  "WAIT_FOR_RELEASE",
  "SET_TIME",
  "INC_TIME",
  "CHOICE",
};

typedef enum
{
  STARTUP,
  WAIT_FOR_RELEASE,
  SET_TIME,
  INC_TIME,
  CHOICE,
} FSM_STATE;

enum Mode
{
  CURRENT,
  ALARM,
  SET,
} ;

struct Time
{
  unsigned char hours;
  unsigned char minutes;
  unsigned char seconds;
};

struct Time current_time, alarm_time;
unsigned char alarm_triggered = 0;
```

```
void DisplayString(BYTE pos, char* text);
void DisplayTop(char* text);
char* current_time_string(enum Mode mode);
void display_state(FSM_STATE state);
void change_time(unsigned char hms, enum Mode mode);
void display_time(enum Mode mode);
void setup(void);
void init_Time(struct Time* time, unsigned char hours, unsigned char minutes, unsigned
     char seconds);

#if defined(__SDCC__)
/**************************************************
 Function DisplayString:
 Writes the first characters of the string in the remaining
 space of the 32 positions LCD, starting at pos
 (does not use strlcopy, so can use up to the 32th place)
**************************************************/
void DisplayString(BYTE pos, char* text)
{
  BYTE        l = strlen(text);/*number of actual chars in the string*/
  BYTE      max = 32 - pos;  /*available space on the lcd*/
  char       *d = (char*)&LCDText[pos];
  const char *s = text;
  size_t      n = (l < max) ? l : max;
  /* Copy as many bytes as will fit */
  if (n != 0)
    while (n-- != 0)*d++ = *s++;
  LCDUpdate();

}

/* Same as DisplayString, but only displays on top row of LCD */
void DisplayTop(char* text)
{
  BYTE        l = strlen(text);/*number of actual chars in the string*/
  char       *d = (char*)&LCDText[0];
  const char *s = text;
  size_t      n = (l < 16) ? l : 16;
  /* Copy as many bytes as will fit */
  unsigned char i = 16 - n;
  if (n != 0)
  {
    while (n-- != 0)*d++ = *s++;
    while (i-- != 0)*d++ = ' ';
  }
  LCDUpdate();
}

#endif
```

```
void high_isr (void) interrupt 1
{
  static unsigned char ticks = 0;
  static unsigned char led_data = 0;
  static unsigned char led_on_time = 0;

  if (INTCONbits.TMROIF) //If TMRO flag is set
  {
    INTCONbits.TMROIE = 0;  // Disable TMRO interrupts
    if ((++ticks % 125 == 0) && alarm_triggered)
    {
      if (led_on_time++ < 60)
        led_data ^= 2;
      else
      {
        led_data = 0;
        led_on_time = 0;
        alarm_triggered = 0;
      }
      LED_PUT(led_data);
    }
    if (ticks == 250)
    {
      current_time.seconds++;
      ticks = 0;
      if (!alarm_triggered)
      {
        if ((current_time.hours == alarm_time.hours) && (current_time.minutes ==
            alarm_time.minutes) && (current_time.seconds == alarm_time.seconds))
          alarm_triggered = 1;
      }
    }
    INTCONbits.TMROIF = 0;  // Reset TMRO flag
    INTCONbits.TMROIE = 1;  // Re-enable TMRO interrupts
    TMROH = HIGH;           // Set TMRO values
    TMROL = LOW;
  }
}

char* current_time_string(enum Mode mode)
{
  char string[12];
  unsigned char i = 8;
  struct Time* time = (mode == ALARM ? &alarm_time : &current_time);

  string[0] = (time->hours / 10) + ’0’;
  string[1] = (time->hours % 10) + ’0’;
  string[2] = ’:’;
  string[3] = (time->minutes / 10) + ’0’;
  string[4] = (time->minutes % 10) + ’0’;
  string[5] = ’.’;
  string[6] = (time->seconds / 10) + ’0’;
  string[7] = (time->seconds % 10) + ’0’;
  for (; i < 12; i++) string[i] = ’ ’;
  return string;
}
```

```c
/* Displays current time on lower row of LCD display */
void display_time(enum Mode mode)
{
  DisplayString(20, current_time_string(mode));
}

/* Displays the current FSM state in the upper row of LCD display. Used in debug. */
void display_state(FSM_STATE state)
{
  DisplayTop(state2str[state]);
}

/* Changes the current or alarm time, depending on the mode. */
void change_time(unsigned char hms, enum Mode mode)
{
  struct Time *time = (mode == ALARM ? &alarm_time : &current_time);

  DelayMs(20);  //arbitrary delay
  if (hms == 1)
  {
    if (mode != ALARM) DisplayTop("Set current hrs");
    else  DisplayTop("Set alarm hours");

    if (BUTTON0_IO == 0u && BUTTON1_IO == 1u)
      time->hours = (time->hours > 0 ? time->hours - 1 : 23);
    else if (BUTTON0_IO == 1u && BUTTON1_IO == 0u)
      time->hours = (time->hours < 23 ? time->hours + 1 : 0);
  }
  else if (hms == 2)
  {
    if (mode != ALARM) DisplayTop("Set current mins");
    else  DisplayTop("Set alarm mins");

    if (BUTTON0_IO == 0u && BUTTON1_IO == 1u)
      time->minutes = (time->minutes > 0 ? time->minutes - 1 : 59);
    else if (BUTTON0_IO == 1u && BUTTON1_IO == 0u)
      time->minutes = (time->minutes < 59 ? time->minutes + 1 : 0);
  }
  else if (hms == 3)
  {
    if (mode != ALARM) DisplayTop("Set current secs");
    else  DisplayTop("Set alarm secs");

    if (BUTTON0_IO == 0u && BUTTON1_IO == 1u)
      time->seconds = (time->seconds > 0 ? time->seconds - 1 : 59);
    else if (BUTTON0_IO == 1u && BUTTON1_IO == 0u)
      time->seconds = (time->seconds < 59 ? time->seconds + 1 : 0);
  }
}
```

```
/* Setup initialization. Run at startup . */
void setup(void)
{
  LED0_TRIS = 0; //configure 1st led pin as output (yellow)
  LED1_TRIS = 0; //configure 2nd led pin as output (red)
  LED2_TRIS = 0; //configure 3rd led pin as output (red)

  BUTTON0_TRIS = 1; //configure button0 as input
  BUTTON1_TRIS = 1; //configure button1 as input

  // TMR0 SETUP
  TMR0H = HIGH;
  TMR0L = LOW;
  T0CONbits.TMR0ON = 0; //stop timer
  T0CONbits.T08BIT = 0;  //16bit
  T0CONbits.T0CS = 0;   //Clock source = instruction cycle CLK
  T0CONbits.T0SE = 0;   //Rising edge
  T0CONbits.PSA = 1;    //No prescaler

  //  INTERRUPT CONFIG
  INTCONbits.GIE = 1;   //enable global interrupts
  INTCONbits.TMR0IE = 0; //enable timer0 interrupts
  INTCON2bits.TMR0IP = 1; //TMR0 has high prio

  LCDInit();
  DelayMs(10);
  LED_PUT(0x00);

  T0CONbits.TMR0ON = 1;  //Enable TMR0

}

/* Initializes instance of struct Time. */
void init_Time(struct Time* time, unsigned char hours, unsigned char minutes, unsigned
     char seconds)
{
  time->hours = hours;
  time->minutes = minutes;
  time->seconds = seconds;
}
```

```
void main(void)
{
  unsigned char hms = 0;
  FSM_STATE state = STARTUP, previous_state = STARTUP;
  enum Mode mode = SET;
  setup();

  while (1) // STATE MACHINE
  {
    //display_state(state); //Display current FSM state (debug)
    display_time(mode);
    if (alarm_triggered)
    {
      DisplayTop("WAKE UP!");
      state = INC_TIME;
    }

    switch (state)
    {
    case (STARTUP):
      init_Time(&current_time, 0, 0, 0);
      init_Time(&alarm_time, 0, 0, 0);

      hms = 1;
      previous_state = STARTUP;
      state = SET_TIME;
      break;

    case (WAIT_FOR_RELEASE):
      if (BUTTON0_IO == 1u && BUTTON1_IO == 1u)
      {
        hms++;
        state = SET_TIME;
      }
      if (hms > 3)
      {
        hms = 0;
        if (previous_state == STARTUP)
        {
          mode = ALARM;
          hms++;
        }
        else
        {
          state = INC_TIME;
          mode = CURRENT;
          INTCONbits.TMR0IE = 1; //enable timer0 interrupts
        }
        previous_state = WAIT_FOR_RELEASE;
      }
      break;
```

```c
    case (SET_TIME):
      INTCONbits.TMR0IE = (mode == SET ? 0 : 1); //disable timer0 interrupts when setting time

      change_time(hms, mode);

      if (BUTTON0_IO == 0u && BUTTON1_IO == 0u)
        state = WAIT_FOR_RELEASE;

      break;

    case (INC_TIME):

      if (current_time.seconds >= 60)
      {
        current_time.seconds = 0;
        if (++current_time.minutes >= 60)
        {
          current_time.minutes = 0;
          if (++current_time.hours >= 24)
            init_Time(&current_time, 0, 0, 0);
        }
      }

      if (!alarm_triggered)
      {
        if (previous_state == CHOICE || (BUTTON0_IO == 0u && BUTTON1_IO == 0u))
        {
          previous_state = state;
          state = CHOICE;
        }
        else
          DisplayTop("Have a nice day!");
      }

      break;

    case (CHOICE):
      previous_state = CHOICE;
      DisplayTop("^Alarm  vCurrent");

      if (BUTTON0_IO == 0u && BUTTON1_IO == 1u)
        mode = SET;
      else if (BUTTON0_IO == 1u && BUTTON1_IO == 0u)
        mode = ALARM;

      while ((BUTTON0_IO == 0u || BUTTON1_IO == 0u) && current_time.seconds != 60); // wait for release

      if (current_time.seconds == 60)
        state = INC_TIME;
      else if (mode == SET || mode == ALARM)
        state = WAIT_FOR_RELEASE;
      break;
    default:
      state = STARTUP;

    } //end switch
  }   //end while
}
```