Alarm Clock
Embedded Systems Project Report

Monday 10:00 lab

Jakub Pawlak
234767@edu.p.lodz.pl

Artur Pietrzak
234768@edu.p.lodz.pl

Juliusz Szymajda
234769@edu.p.lodz.pl

June 5, 2022

Devices used:

Eduboard LPC2148 v1.0

No external devices used

Interfaces used:

GPIO, I^2C, SPI

Devices used:

1. LCD display
2. RTC
3. Button
4. Joystick
5. Buzzer
6. Timer
7. EEPROM

Contents

Code Listings

1 Project Description

1.1 General description

The project is a digital clock with the ability to set an alarm for a given hour. The device displays current time on the LCD display. Using button and joystick it is possible to set the current time, or set an alarm. If the alarm is set and the alarm time is the current time, the buzzer emits a sound until the alarm is turned off.

1.2 Project functionalities

Functionality	**Person Responsible**	**Implementation status**
Timer	Jakub Pawlak	Implemented
RTC	Artur Pietrzak	Implemented
LCD Display	Jakub Pawlak	Implemented
Button & Joystick	Artur Pietrzak	Implemented
Buzzer	Jakub Pawlak	Implemented
EEPROM	Juliusz Szymajda	Implemented
I2C	Juliusz Szymajda	Implemented

Table 1: Project functionalities and responsible persons

1.3 Setting the alarm

The alarm setting mode is entered by holding down the button. When in alarm setting mode, the user can change the alarm time by selecting the digit with left/right movement, and selecting the value with up/down movement. After the correct time is selected, alarm is set by pressing the button.

When alarm is turned on, it can be turned off by short press of the button.

1.4 Changing the current time

The time setting mode is entered by pressing the button for a short time. Then, the time is set with the joystick in the similar way, the alarm is set. After setting the time it is set by short press of the button.

1.5 Turning off the alarm

When the alarm clock starts emmiting sound, the user can turn it off by pressing the button.

2 Peripherals and interface configuration

2.1 GPIO

Example of GPIO usage in our project is for the buzzer. The board schematic shows that for this purpose we must use P0.7 [1, p.9].

```
#define ALARM_PIN 7

static void init_buzzer()
{
    PINSEL0 &= ~(3 << (2 * ALARM_PIN)); // clear bits 15:14

    IODIR0  |= (1 << ALARM_PIN); // set P0.7 as output

    IOSET0  =  (1 << ALARM_PIN); // set P0.7 high (buzzer off)
}

static void buzzer_on()
{
    IOCLR0 = (1 << ALARM_PIN); // bring P0.7 low
}

static void buzzer_off()
{
    IOSET0 = (1 << ALARM_PIN); // set P0.7 high
}
```

Listing 1: Code used for controlling the buzzer pin

The pin we want to use can have many functionalities, so the first thing that needs to be done, is to set it as GPIO port. This is accomplished by writing to the PINSEL register. The bits 15:14 control port P0.7, and value 0 corresponds to GPIO functionality [2, p.59].

Next, we need to set the port to function as output. This is done via the IODIR register, so we set the 7th bit to 1, which means that P0.7 will be used as output.

Finally, we write the 7th bit to IOSET register, to set the pin high. This is done to turn off the buzzer, so it does not make noise from the start.

From the diagram [1, p.9], we can see that the buzzer is controlled via PNP transistor, so we turn it on by setting the pin low, and vice versa. To set the pin low, we write the appropriate bit to IOCLR register, and to set it high, we write to IOSET register.

It is also possible to control the pin state directly via IOPIN register, however working with IOSET and IOCLR is easier, because it eliminates the need to think about logic operations necessary to avoid accidentally changing the state of other pins.

2.2 LCD Display

2.2.1 Setup

We begin setting up the lcd by configuring the IODIR registers to set the appropriate pins as output and clear them. The pins used are data pins P1.6 to P1.3, represented as LCD_DATA, lcd enable pin P1.5 (LCD_E), lcd read/write pin P0.2 (LCD_RW), register select P1.4 (LCD_RS), and backlight P0.0 (LCD_BACKLIGHT).

Next, we proceed by sending a command $(38)_{16}$, which represents the 'Function Set' instruction. We set the data length to 8 bits, and the number of display lines to 2.

Next, we send display on/off command, to reset the display.

Then, we clear the display and send the $(06)_{16}$ command, which sets the cursor direction to 'increment'.

Finally, we turn on the display, without cursor, and set cursor to home position.

All of the values to send, were taken from the instruction table in the display driver manual [3, p.24]

2.2.2 Sending data to the display

The core of interaction with the LCD is the `writeLCD` function, which takes 2 parameters - register (0 — instruction or 1 — data), and the actual 8-bit data we want to send. We begin by setting the register select pin (lines 6–9).

Before we send any data, we clear the read/write pin, to indicate that we want to write data to the display. Then we clear the data pins, and set the value in line 13. We first have to shift the value by 16 bits to the left, because we use pins P1.6–P1.3, and then we mask them with LCD_DATA to make sure that we do not set any other pins.

```
void DisplayInit(void) {
    IODIR1 |= (LCD_DATA | LCD_E | LCD_RS);
    IOCLR1  = (LCD_DATA | LCD_E | LCD_RS);

    IODIR0 |= LCD_RW;
    IOCLR0  = LCD_RW;

    IODIR0 |= LCD_BACKLIGHT;
    IOCLR0  = LCD_BACKLIGHT;

    LcdCommand(0x30);
    delay2ms();
    LcdCommand(0x30);
    delay37us();
    LcdCommand(0x30);
    delay37us();

    LcdCommand(0x38); // set 8-bit, 2 line mode
    delay37us();

    LcdCommand(0x08); // display off
    delay37us();

    clearDisplay();

    LcdCommand(0x06); // cursor direction - increment, no shift
    delay2ms();

    LcdCommand(0x0c); // display on, cursor off
    delay2ms();

    LcdCommand(0x02); // cursor to home position
    delay2ms();
}
```

Listing 2: LCD setup function

After setting the data pins, we set the LCD_E pin, which starts the data read. After some delay, we stop the data transmission by clearing LCD_E pin and wait an additional delay before the next piece of data can be sent.

2.2.3 Moving the cursor

The text that is currently displayed on the LCD is stored in so-called 'display data RAM' (DDRAM). The cursor is located at whatever address is currently stored at the DDRAM address counter [3, p.21].

Therefore, we can change the cursor position, by changing the address in the AC. Luckily, the LCD controller has an instruction for just that purpose. It is used by sending a command with bit 7 set, followed by 7-bit DDRAM address.

The first row is addresses $(00)_{16}$ – $(27)_{16}$, and the second row is addresses $(40)_{16}$ – $(67)_{16}$ [3, p.11].

Therefore, for setting the cursor position, we created the following function:

```
static void writeLCD(uint8 reg, uint8 data)
{
    volatile uint8 i;

    if (reg == 0)
      IOCLR1 = LCD_RS;
    else
      IOSET1 = LCD_RS;

    IOCLR0 = LCD_RW;
    IOCLR1 = LCD_DATA;
    IOSET1 = ((uint32)data << 16) & LCD_DATA;

    IOSET1 = LCD_E;

    for(i=0; i<16; i++)
        asm volatile (" nop"); //15ns x 16 = about 250 ns
    IOCLR1 = LCD_E;

    for(i=0; i<16; i++)
        asm volatile (" nop"); //15ns x 16 = about 250 ns
    delay2ms();
}

void LcdCommand(unsigned char byte){
    writeLCD(0, byte);
}

void LcdData(unsigned char byte){
    writeLCD(1, byte);
}

void clearDisplay() {
    LcdCommand(0x01);
    delay2ms();
}
```

Listing 3: Functions for using the display

```
void SetCursor(unsigned int line, unsigned int column){
    if (column > 39)
        column = 39;

    if (line == 0)
        LcdCommand(0x80 + column);
    else
        LcdCommand(0xc0 + column);
}
```

Listing 4: Function for controlling the cursor position

Firstly, we limit the column to not be greater than 39, as the DDRAM only has 40 characters per line.

Then, we send the 'Set DDRAM address' command with the appropriate address. If we want to set the position in the first line, we send $(80)_{16}$ (7-th bit set, indicating command) followed by the column, as the 1st line address starts from 0. If we want to set the position in the second line, instead of $(80)_{16}$ we use $(C0)_{16}$, which is $(80)_{16}$ (7th bit) + $(40)_{16}$ (starting address of the 2nd line).

2.2.4 Writing text

Writing text to display is very simple, we just send data, and the cursor automatically increments position after each character. We just loop over the array of characters and write each one using the `LcdData` function from listing 3.

```
void LcdPrint(char* str){
    while(*str){
        LcdData(*str);
        delay37us();
        str++;
    }
}
```

Listing 5: Function for writing text to the display

2.3 Joystick

```
char getJoyInput(void) {
    if (((IOPIN0 & KEYPIN_UP) == 0)) {
        return 'u';
    }
    if (((IOPIN0 & KEYPIN_DOWN) == 0)) {
        return 'd';
    }
    if (((IOPIN0 & KEYPIN_RIGHT) == 0)) {
        return 'r';
    }
    if (((IOPIN0 & KEYPIN_LEFT) == 0)) {
        return 'l';
    }
    if (((IOPIN0 & KEYPIN_CENTER) == 0)) {
        return 'c';
    }
    return 0;
}
```

Listing 6: Joystick get input function

The function checks if the joystick is utilized and return char corresponding to the current position of the joystick.

```
char waitForJoyInput(void){
    while (1) {
        char result = getJoyInput();
        if (result != 0)
            return result;
        delay37us();
    }
}
```

Listing 7: Await joystick input function

This function runs in an infinite loop, which ends when the result of the JoyInputFunction is not 0 (when there is some input from the joystick).

```
uint8 isButtonPressed(void) {
    if ((IOPIN0 & PIN_BUTTON) == 0) {
        return TRUE;
    }
    else {
        return FALSE;
    }
}
```

Listing 8: Get button input function

This function check wether the button on the board is pressed.

2.4 RTC

2.4.1 Initialization

```
RTC_Time setTime, alarmTime, currentTime;
ILR = 0x0;
CCR = 0x02;
CCR = 0x00;
CIIR = 0x00;
AMR = 0x00;
PREINT = 2000;
PREFRAC = 50;
CIIR = 0x00;
CCR = 0x01;
```

Listing 9: RTC initialization

First, the structs for time to set, alarm time and current time are created. The RTC interrupts are disabled by writing 0x0 to Interrupt Location Register (ILR). The clocks tick counter is also reset by writing 0x02 to the Clock Control Register (CCR), and disabled by writing 0x0. Next, the Counter Increment Interrupt (CIIR), which is responsible for incrementing seconds, is disabled.

Next, the prescaler is configured. Prescaler is a circuit used to reduce a high frequency electrical signal (for example from RTC) to a lower frequency by integer division. After calculating the value of a prescaler integer:

$$\left(\frac{\text{PCLK}}{32768}\right) - 1$$

and value of prescaler fraction:

$$\text{PCLK} - ((\text{PREINT} + 1) \cdot 32768)$$

Those values are written to Prescaler Integer Register (PREINT) and Prescaler Fraction Register (PREFRAC) registers.

Finally, the time is read from EEPROM, written into setTime struct and set using RTC_SetTime function. At the end of initialization 0x01 is writtent to Clock Control Register (CCR), which enables the timer counters.

```
typedef struct
{
    uint8 seconds;
    uint8 minutes;
    uint8 hours;
}
```

Listing 10: RTC struct

The struct consists of three 8 bit unsigned integer numbers, each one for seconds, minutes and hours fields.

2.4.2 Setting the time

```
void RTC_SetTime(RTC_Time Time)
{
    SEC = Time.seconds;
    MIN = Time.minutes;
    HOUR = Time.hours;
}
```

Listing 11: Setting time in the register function

The function uses data stored in the parsed struct and writes it to counters from Time Register Group. SEC and MIN are 6 bit, and the HOUR counter is 5 bit.

```
void RTC_Set_AlarmTime(RTC_Time AlarmTime)
{
    ALSEC = AlarmTime.seconds;
    ALMIN = AlarmTime.minutes;
    ALHOUR = AlarmTime.hours;
}
```

Listing 12: Setting alarm time in the register function

The function uses data stored in the parsed struct and writes it to counters from Alarm Register Group. Similarly to the Time Register Group, ALSEC and ALMIN are 6 bit, and the ALHOUR counter is 5 bit.

2.4.3 Reading the time

```
RTC_Time RTC_GetTime(void)
{
    RTC_Time time;

    time.seconds = SEC;
    time.minutes = MIN;
    time.hours = HOUR;

    return time;
}
```

Listing 13: Getting time from the register function

The function takes data from SEC, MIN and HOUR registers stored in Time Register Group, assigns it to the newly created RTC_Time struct and returns it.

```
RTC_Time RTC_GetAlarmTime(void)
{
    RTC_Time AlarmTime;

    AlarmTime.seconds = ALSEC;
    AlarmTime.minutes = ALMIN;
    AlarmTime.hours = ALHOUR;

    return AlarmTime;
}
```

Listing 14: Getting alarm time from the register function

Analogously, to the RTC_GetTime function, this function takes data from ALSEC, ALMIN and ALHOUR registers stored in Alarm Register Group, assigns it to the newly created RTC_Time struct and returns it.

```
bool RTC_Compare(RTC_Time t1, RTC_Time t2)
{
    return (t1.seconds == t2.seconds
        && t1.minutes == t2.minutes
        && t1.hours == t2.hours);
}
```

Listing 15: RTC_Time struct comparator function

The function is used to compare two RTC_Time structs by comparing each field of the struct (seconds, minutes and hours).

2.5 Timer

For producing precise delay we use timer 0 of the microcontroller.

To start, we reset and disable the timer, TIMER_RESET to the timer control register (TCR). Then, we can write the values controlling the delay. Firstly we write the prescaler to prescale counter (PC). We have chosen the value $12000000-1$, becuse the clock runs at 12 MHz. Then, we write actual delay length to the match register 0 (MR0).

Then, we write MR0_S to match control register (MCR) to indicate that we want the timer to stop at MR0 match. We write TIMER_RUN to the control register to enable the timer and start counting [2, p.251].

Finally, we enter a loop to wait until the timer is running.

```
1  static void sdelay (tU32 seconds)
2  {
3      T0TCR = TIMER_RESET;
4      T0PR  = 12000000-1;
5      T0MR0 = seconds;
6      T0IR  = TIMER_ALL_INT;
7      T0MCR = MR0_S;
8      T0TCR = TIMER_RUN;
9
10     while (T0TCR & TIMER_RUN)
11     {
12     }
13 }
```

Listing 16: Timer delay function

2.6 EEPROM

We use I2C to save in EEPROM the alarm time chosen by the user. Then, the alarm time will be read in order to compare it with the real time. To initialise I2C, we set pins P0.02 and P0.03 to value 01 with operation:

```
PINSEL0 |= 0x50;
```

Next, the flags are cleared:

```
I2C_CONCLR = 0x6c;
```

Then we set registers with following operations:

```
I2C_SCLL   = ( I2C_SCLL   & ~I2C_REG_SCLL_MASK )   | I2C_REG_SCLL;
I2C_SCLH   = ( I2C_SCLH   & ~I2C_REG_SCLH_MASK )   | I2C_REG_SCLH;
I2C_ADDR   = ( I2C_ADDR   & ~I2C_REG_ADDR_MASK )   | I2C_REG_ADDR;
I2C_CONSET = ( I2C_CONSET & ~I2C_REG_CONSET_MASK ) | I2C_REG_CONSET;
```

All of the above operations are collected in the function "i2cInit".

EEPROM (electrically erasable programmable read-only memory) is used to save the alarm time chosen by user. We operate it using I2C interface. To save the time to memory we initialise I2C and then use the function "eepromWrite" that has 3 parameters:

1. Hexadecimal address from which data is to be written to the memory (In our case always 0x0000)
2. Type of data saved (array of chars)
3. Number of characters including the address (length)

As the address and length is always the same, we created second function ("save") that only takes one argument, which is data to be saved. To read from the memory, we also have to initialise I2C and use function "eepromPageRead" which has 3 identical parameters as the "eepromWrite". To simplify the use, we created second function ("read") that only takes one argument, which is data "container" to which we want to read the time. Both functions stop I2C before terminating.

```
tS8 eepromPageRead(tU16 address, tU8* pBuf, tU16 len)
{

tS8  retCode = 0;
tU8  status  = 0;
tU16 i       = 0;

/* Write 4 bytes, see 24C256 Random Read */
retCode = eepromStartRead(I2C_EEPROM_ADDR, address);


if( retCode == I2C_CODE_OK )
{
 /* wait until address transmitted and receive data */
 for(i = 1; i <= len; i++ )
 {
     /* wait until data transmitted */
     while(1)
     {
         /* Get new status */
         status = i2cCheckStatus();

         if(( status == 0x40 ) || ( status == 0x48 ) || ( status == 0x50 ))
         {
             /* Data received */

             if(i == len )
             {
                 /* Set generate NACK */
                 retCode = i2cGetChar( I2C_MODE_ACK1, pBuf );
             }
             else
             {
                 retCode = i2cGetChar( I2C_MODE_ACK0, pBuf );
             }

             /* Read data */
             retCode = i2cGetChar( I2C_MODE_READ, pBuf );
             while( retCode == I2C_CODE_EMPTY )
             {
                 retCode = i2cGetChar( I2C_MODE_READ, pBuf );
             }
             pBuf++;

             break;
         }
         else if( status != 0xf8 )
         {
             /* ERROR */
             i = len;
             retCode = I2C_CODE_ERROR;
             break;
         }
     }
 }
}

/* Generate Stop condition */
i2cStop();

return retCode;

}
```

Listing 17: EEPROM page read function

```
tS8 eepromWrite(tU16 addr, tU8* pData, tU16 len)
{

tS8 retCode = 0;
tU8 i       = 0;

do
{

    /* generate Start condition */
    retCode = i2cStart();
    if(retCode != I2C_CODE_OK)
        break;


    /* write EEPROM I2C address */
    retCode = i2cWriteWithWait(I2C_EEPROM_ADDR);
    if(retCode != I2C_CODE_OK)
        break;

    /* write offset low in EEPROM space  */
    retCode = i2cWriteWithWait( (tU8)(addr & 0xFF));
    if(retCode != I2C_CODE_OK)
        break;

    /* write data */
    for(i = 0; i <len; i++)
    {
        retCode = i2cWriteWithWait(*pData);
        if(retCode != I2C_CODE_OK)
            break;

        pData++;
    }

} while(0);

/* generate Stop condition */
i2cStop();


return retCode;
}
```

Listing 18: EEPROM write function

```
void save(tU8 string[]){
    eepromWrite(0x0000, string, 2);
}

void read(tU8 string[]){
    eepromPageRead(0x0000, string, 2);
}
```

Listing 19: Helper functions for EEPROM

3 Failure Mode and Effect Analysis

3.1 Failure severity

The project depends on its components to be able to function properly. However, not all of them are equally important — some are necessary for the usage of the project, while others just provide additional functionalities.

The table bellow presents a list of components considered in failure effect analysis, as well as the significance of it's failure.

Component	**Severity**
Microcontroller	Critical
Power Supply	Critical[1]
RTC	Critical
LCD Display	High
Buzzer	Medium
Button	Medium
Joystick	Medium
EEPROM	Low

Table 2: Severity of component's failure

The most crucial elements of the project are the microcontroller, power supply, and RTC. The microcontroller must not fail, because it is responsible for all of the logic of the device. The power supply must also be functioning, because there is no backup battery able to sustain the device operation. The RTC is necessary for the operation of the project, because it ensures the correctness of the told time.

The failure of the LCD display is of high severity, because this is the main way of interacting with the clock. However, it is not of critical severity, because even without the display, if the alarm was set, it will ring at the specified time.

The buzzer, button and joystick are used for the alarm, which is the additional functionality of the clock. Without them it is still possible to use the device for telling the current time.

The EEPROM failure is of the low severity, because it is only used for backup storage in case of short power supply failure. In the normal circumstances, it's failure would not at all impact the usability of the device.

[1] Long-term power supply failures are of critical severity, but in case of short pause in power delivery, the system is able to recover using the data stored in EEPROM

3.2 Failure detection

The failure of the microcontroller would be very easily noticeable, as it would probably result in a complete failure of the device.

The lack of power is also easily detectable, because the device will not show anything. The board is equipped with a small green power indicator LED [1]. Should that indicator not be lit, means the failure of the power supply.

The LCD failure is also trivial to notice, as the display would not be showing anything. To differentiate it from the power supply failure, the power indicator LED should be inspected.

References

[1] Embedded Artists. *LPC2148 Education Board User's Guide*, 2006. EA2-USG-0601 v1.2 Rev B.

[2] NXP. *LPC214x User manual*. Rev. 4 — 23 April 2012.

[3] Hitachi. *HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/-Driver)*. ADE-207-272(Z) '99.9 Rev. 0.0.