Custom-Built Desk Clock with Environmental Sensing

Author: VLAD Gabriel-Lucian

University: "*National University of Science and Technology POLITEHNICA Bucharest* "

Table of Contents

1. Introduction

This section provides an overview of the Custom-Built Clock with Environmental Sensing project.

1.1. Problem Statement

Being an Electronic Engineering student living in the student dormitory I encountered a lot of waking up in the middle of the night not being able to tell what time it is without using my phone. I started to think of ways to solve this problem and came up with a few ideas, one of them being this project. It was a convenient solution since I already had some spare electronic parts from other projects.

Even though I could just buy some cheap digital clock, I started to develop my own device since this project is not motivated by cost savings, but rather to extend my knowledge and experience on how to design a device to fit certain needs.

1.2. Objective

The objective of this project is to design and implement a portable desk clock capable of displaying both time and environmental information. This project aims to integrate micro-controllers, sensors, user interaction elements and display units, all of them encapsulated in a battery-powered device.

2. Device Operation

2.1. Powering The Device

Being a **battery-powered** device, special care had to be taken regarding its power source. So, I figured that adding a **ON/OFF switch** to it would make the device last longer before it needs a battery replacement. The switch is on top of the device, when " **O** " position is pressed the battery stops providing energy to the micro-controller, shutting it down. When " **I** " is pressed, the battery is connected to the " **RAW** " **pin** of the micro-controller, powering the device and making it fully functional.

2.2. Normal Operation

After the device is turned on and the micro-controller does its **setup routine**, the **LCD starts displaying** the current date and time on the first row, and on the second row the temperature.

2.3. Setting The Date and Time

Once the device is running, the date and time can be modified if the " **SET** " **button** is pressed. Then it will enter in the "**Setting Mode**" and the users is prompted to **set** each **parameter** (day, month, year, weekday, hour and minute) of the date and time one by one by **pressing the other two buttons:** "**INC**" **button** (for increasing the value to be set) or "**DEC**" **button** (for decreasing the value to be set). To jump to the **next parameter** the user needs to **press** the "**SET**" button **again**. **After** all the **parameters** are **set** by the user, when the "**SET**" button is pressed one last time the device goes back into its "**Normal Operation**" state.

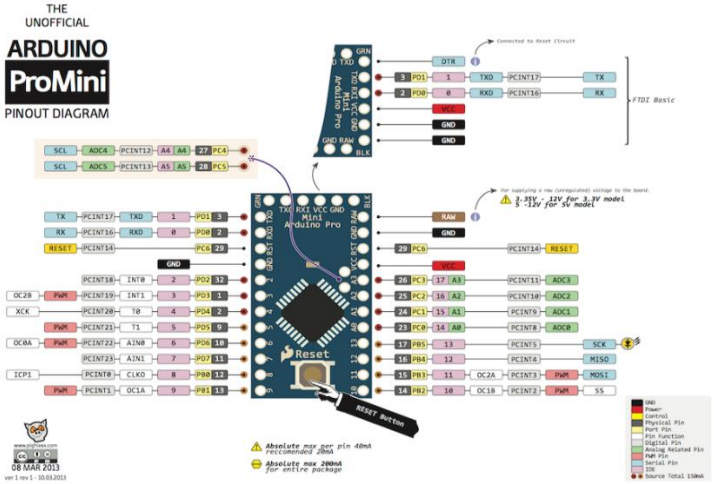
3. Theoretical Background

3.1. Presentation of Key Concepts

In the following subsection, I will briefly present the main electronic components used in the project, along with their role and relevant technical data: Arduino Pro Mini 5V (ATmega328P 8-bit microcontroller), LCD 1602, DS1307 RTC module, DHT-11 temperature and humidity, a 9V battery, sensor, resistors, push buttons and a 6F22 connector (for 9V battery).

- **Arduino Pro Mini 5V 16MHz, ATmega328P microcontroller**

It is a microcontroller board based on the **ATmega328P AVR Architecture,** which is a reduced instruction set computer (**RISC**) architecture. It has **14 digital input/output pins**, **6 analog inputs**, an on-board **resonator** and a **reset button**. The microcontroller board is the "**brain**" of the project and all the important information, either it is an input or an output, is managed by the microcontroller unit (**MCU**). The ATmega328P has 32 general-purpose registers and **32kB of flash memory** for storing code, which is more than enough for my project. The Arduino Pro Mini board also comes with a "**RAW**" input voltage pin which offers the possibility to supply the board a higher voltage (**5V – 12V**) since this pin its connected to a **voltage regulator** that outputs the **5V** working voltage for the microcontroller.



Arduino Pro Mini Pinout Diagram

- **LCD 1602**

LCD1602 is a kind of dot matrix module used for **displaying of ASCII characters**. It is composed of 5x7 or 5x11 dot matrix positions; each position can display one character. There's a dot pitch between two characters and a space between lines, thus separating characters and lines. The model **1602** means it displays **2 lines of 16 characters**. In my project I used the 5V version of the LCD because the MCU is also running on logic 5V. In this project, the LCD module is used to display the time, date and temperature values in real time.

- **DS1307 RTC Module**

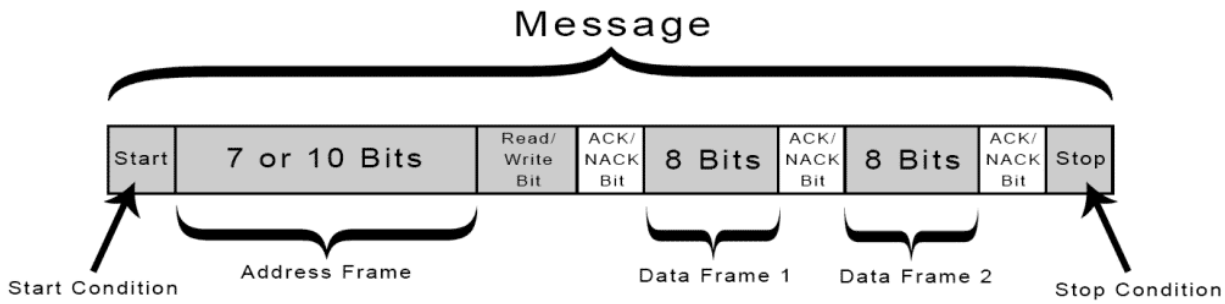
The DS1307 Serial Real-Time Clock is a **low-power clock/calendar** module used for storing and providing seconds, minutes, hours, day, date, month and year information. The end of the month date is automatically adjusted for each month, including corrections for leap years. It stores data using an **AT24C32 EEPROM** Serial 32kB I2C. The DS1307 also comes with the possibility to power it from a CR2032 3V battery or a LIR2303 rechargeable Lithium. The IC has a built-in **power sense circuit** that detects power failures and automatically switches to the **battery** if necessary. The module uses the **I2C** protocol to communicate with the MCU, making it easy to implement in any project. The RTC module ensures accurate timekeeping even when the main power supply is disconnected, by using its backup battery.

- **DHT11**

The DHT11 features a **temperature and humidity sensor** complex with a calibrated digital signal output that uses the exclusive digital-signal-acquisition technique and temperature and humidity sensing technology. This sensor includes a resistive-type humidity measurement component and an **NTC** temperature measurement component. DHT11 communicates with the MCU using **Serial Interface** (Single-Wire Two-Way). **Single-bus data format** is used for communication and synchronization between the MCU and the sensor. The communication process can be summarized as: when MCU sends a **start signal**, DHT11 changes from the **low-power consumption mode** to the **running-mode**, after which DHT11 sends a response signal of **40-bit data** that includes the relative humidity and temperature information to the MCU. **After** the 40-bit data has been sent, DHT11 goes back into **low-power consumption mode**.

- **I^2C communication protocol**

I^2C is a widely used protocol for exchanging data over short distances. In this protocol you can connect **multiple slaves** to a **single master** and you can have multiple masters controlling single, or multiple slaves. It consists of two wires: **SDA** and **SCL**; the **SDA wire** is for master and slave to **send and receive data** and the **SCL** wire is the line that carries the **clock signal**. I2C is a **serial communication** protocol, so data is transferred bit by bit along a single wire, the SDA line. With I2C, data is transferred into **messages** which are broken up into **frames of data**. The message consists of the following: the **start bit** (the SDA line switches to a low voltage level before the SCL line switches from high to low), the **address frame**, the **Read/Write bit** (R/W bit is set by the master either to low voltage level for sending data or high voltage level for requesting data), **data frames of 8 bits**, **acknowledge/no-acknowledge bits** (ACK/NACK) are returned to the sender after each data frame was successfully received or not, finally, after the data has been transmitted a **stop bit** is sent to the sender (the SDA line switches from low to high after the SCL line switches from low to high). The **data sampling** from the SDA line is done during the **middle of the clock pulse**, therefore SDA only transitions when the clock is low.



I2C Message Anatomy

3.2. Technologies and Tools Used

For the development of this project, I used the **Arduino IDE** environment for writing, compiling and uploading **C/C++ code** to the Arduino Pro Mini Board. I chose Arduino IDE because of its large community and the availability of *libraries* for the modules used in this device. Having access to these libraries sped up the development process.

I also used **Autodesk Fusion 360** for creating the device's **circuit diagram** and to design the **custom enclosure** for the desk clock.

3.3. Relevant Examples or Case Studies

Considering that this was more of a **didactical project** there is no point in comparing the total production cost of the device. At the moment of writing this document, the average price for a simple battery-powered desk clock is **around $20**, which is not far cheaper than my prototype. Some key features that these commercial clocks have is that they have automated day/night brightness control and alarms. The battery life for these types of clocks is also longer given the fact that they may not work using an MCU, but rather an ASIC board.

Nevertheless, building this device provided a better understanding of how MCUs, sensors and peripheral modules can be **integrated** into a **fully functional and autonomous system**, even if commercial products are more optimized for cost and power consumption.

4. Requirements and Specifications

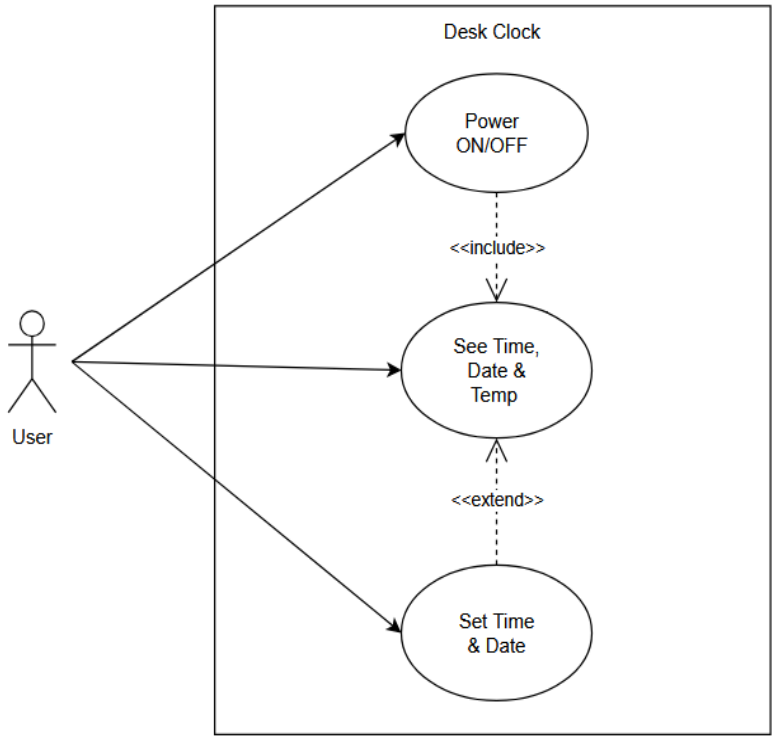
4.1. Functional Requirements

Starting from the idea this device was designed to serve as a desk clock it should've been able to do the following: **display the time and date** while also **storing** this data and the user being able to **update** it to any desired value, **display the ambient temperature**, have a **high portability** and to be **fully autonomous** (powered by a battery).

4.2. Non-Functional Requirements

After assessing the functional requirements, I had to think about how I want the device to encapsulate all those requirements, so these are some key features I implemented; Since it is intended to be powered by a 9V battery a **low-power consumption mode** was a solid option for increasing the battery life. Knowing the clock needed high portability I had to develop a **friendly user interface** which is intuitive; therefore, I added the 3 **buttons** used for **setting** the time and date and a special "**Setting Mode**" that displays each setting step, so the user doesn't get confused. Finally, when designing the device's **enclosure**, I had to keep in mind that it was going to be placed on a desk, where sometimes the **space is limited**, so I came up with a **small rectangular enclosure** just big enough to fit all the components.

4.3. Use Case Diagram



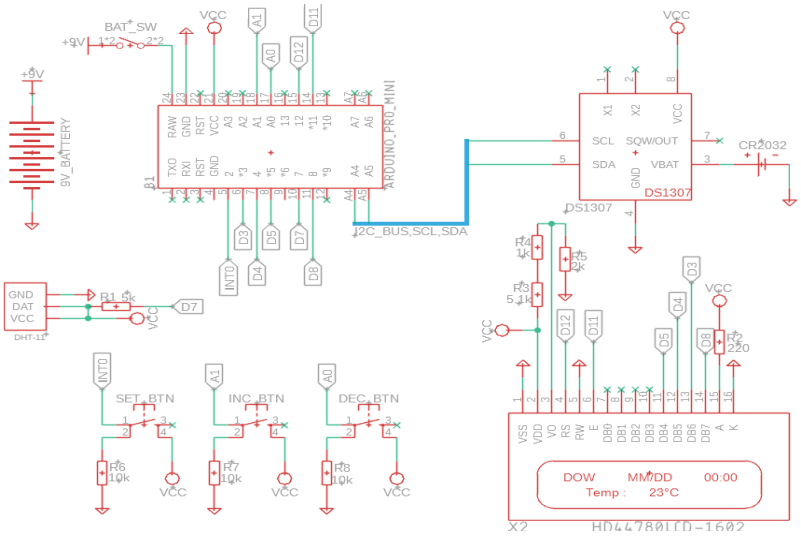
Use case diagram

5. Implementation

5.1. Hardware Implementation

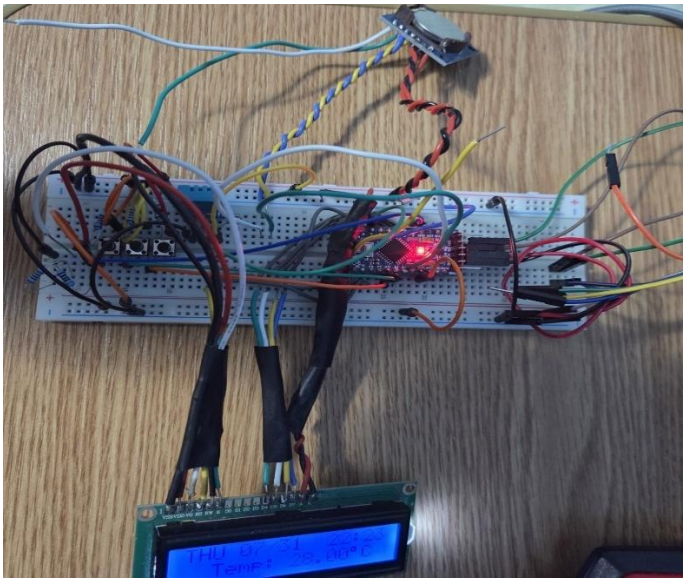
In this following section I'm going to present the hardware side of the project, including the circuit diagram, additional information based on this diagram and details about the custom-built enclosure.

- **Circuit Diagram**



Circuit Diagram

Based on this diagram I've implemented the circuit on a breadboard to be able to test and verify all the functionalities.



Circuit On Breadboard

- **Additional Information**

As can be seen in the above figure, the device is powered by a **9V battery** via a 6F22 connector. This battery was connected to the "**RAW**" pin of the Arduino board, which can sustain any voltage between 5V and 12V. An **"ON/OFF" Switch** was added on this connection route to **save battery life** when the device is not used.

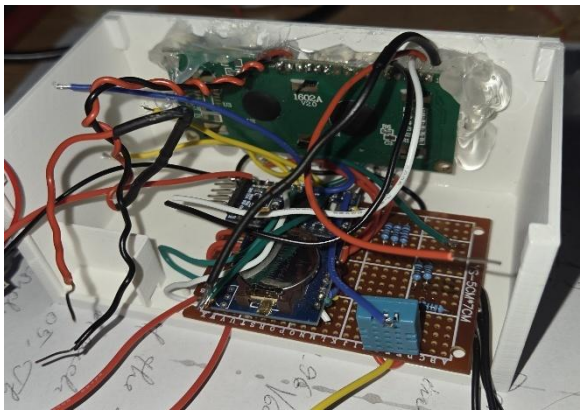
Since the **DS1307 module** requires I2C protocol I have connected **SDA** (Serial Data) and **SCL** (Serial Clock) pins of the module to the **A4 (SDA)** and **A5 (SCL)** pins of the Arduino Board, forming an **I2C Bus**. Also, the DS1307 module supports an additional **CR2032 battery** that is automatically used when the power from VCC is cut off; this battery ensures that the **module keeps the time and date** and stores it into the AT24C32 EEPROM storage.

The **DHT11** communicates with the **MCU** via Digital Pin 7 (**D7**) of the Arduino board, and it has a **5kΩ pull-up resistor** (as recommended by manufacturer in datasheet) to make sure that the **signal level stays high** by default, since the DHT11 waits for the MCU to send the start signal, which means pulling down voltage for at least 18ms.

The **LCD1602** is connected to the **MCU** in a **7-pin configuration (4-bit mode)**. **DB4-DB7** LCD's pins are connected to **digital pins** of **MCU** and are used for reading and writing data. "**A**" (LED+) pin is connected to +5V through a 220Ω resistor while "**K**" (LED-) is connected to GND. The Enable ("**E**") pin is connected to a **digital pin 11** of the **MCU** and the Register Select ("**RS**") pin is connected to **digital pin 12** of the **MCU**, this Register Select pin selects either the data register, which stores what goes on the screen, or an instruction register, which is where the LCD's controller looks for instructions on what to do next (ex: clear, scroll, set cursor, etc.). The **Read/Write pin** is rooted to **GND** since **R/W = 0** is used for **register writing** and in my project, I only write on the LCD, so the R/W pin should always be 0. "**VDD**" pin is the power supply for the logic circuit, therefore it needs to be connected to +5V. Finally, via the "**V0**" pin is used for adjusting the **backlight contrast**. This pin requires a different value than +5V, so a simple solution for supplying a different voltage is to implement a **voltage divider**. At first, for testing reasons, I've used a potentiometer, then I've proceeded to use standardized resistors. Using the basic voltage divider formula ($V0 = \frac{R5}{R3+R4+R5}$), we obtain that **V0 ≈ 0.25V**.

The **three UI buttons** are connected to **analog and digital pins** of the **MCU**. From the diagram we can observe that the pins of the MCU are rooted through GND using **10kΩ pull-down resistors**; so, in the default state of the button we read "**LOW**" because of the pull-down resistor, without it, the input would be "**floating**", and its state would be **unpredictable**. When the **button is pressed** a connection between the pin of the **MCU** and **VCC** is made and we read "**HIGH**". As can be seen in the diagram, the "**SET**" button is not connected to a regular digital pin, it is connected to the "**INT0**" (External Interrupt 0) pin of the **MCU**, as it **triggers an external interrupt** in the code ran by the MCU.

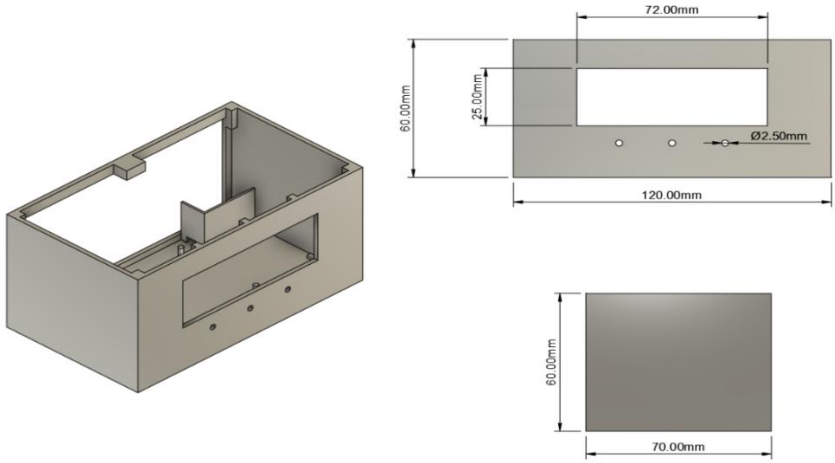
- **Final Assembled Circuit**



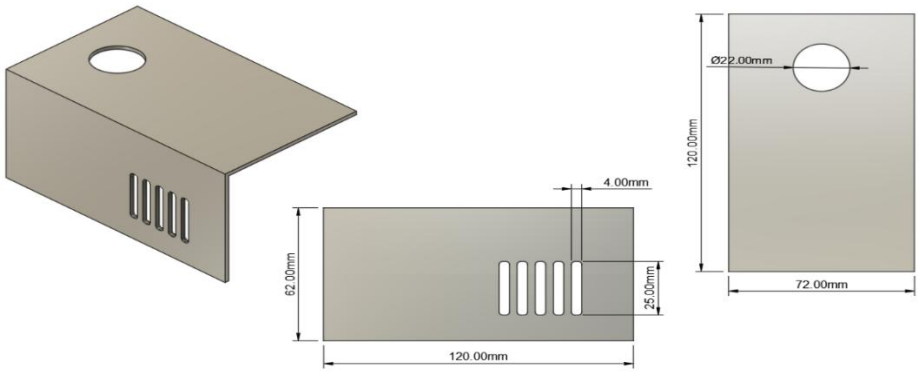
After the development phase of the project, I soldered all the components on a perfboard that will pe placed inside the custom-built enclosure.

- **Custom Enclosure Design**

The custom-built enclosure was designed in Autodesk Fusion 360 software, and it was built to be just big enough to fit all the components and leave some space for any maintenance work if needed. In the following figures there can be observed the dimensions of the enclosure.



Main Enclosure



Enclosure Gasket

5.2. Software Implementation

5.2.1. Code Structure

The code is structured into several main sections:

- Library Inclusion
- Pin Definition
- Global Variables and Constants
- Additional Functions (they help to keep the loop() function cleaner)
- Setup() function
- Loop() function

5.2.2. Libraries Used

One of the libraries used is **LiquidCrystal.h** by Arduino Adafruit, which I used for working with the LCD. Some of the main functions I've used are: **LiquidCrystal()** (for defining the pins used for the LCD and whether it is configurated into 4 or 8 data pins), **print()**, **setCursor()** and **clear()**. The last 3 functions are used for displaying characters on the LCD.

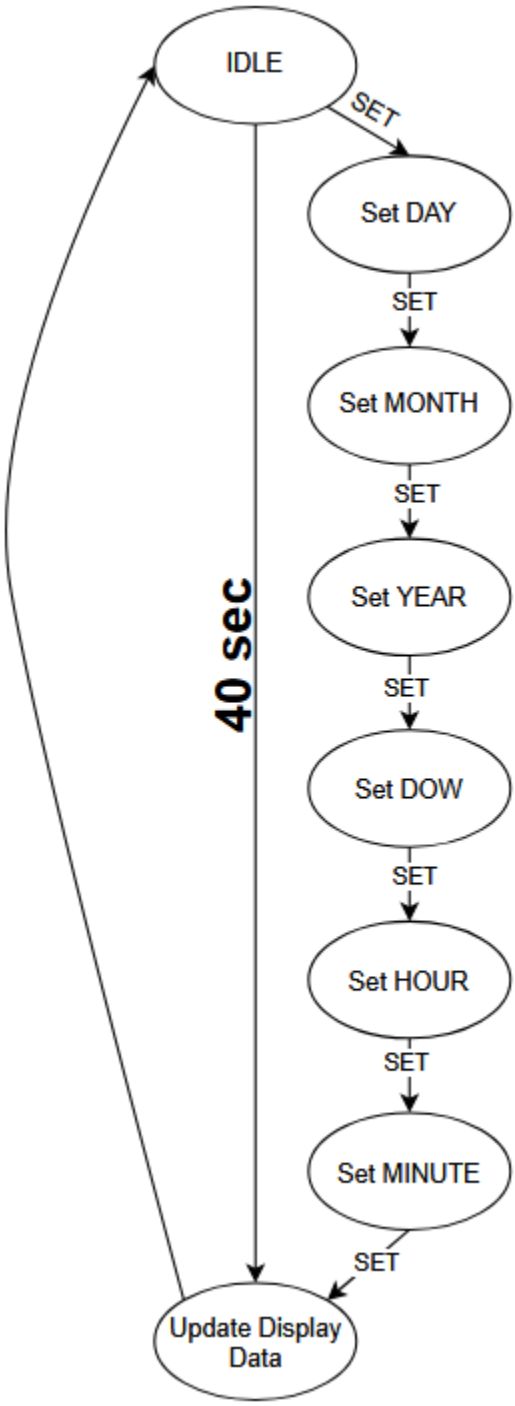
Another library which helped me working with the DHT11 sensor is **DHTlib.h** by Rob Tillaart. It works by defining a **constructor** and then using a read function to communicate with the sensor.

The last library included in the project is **uRTCLib.h** by Naguissa. This library was used to help me work with the RTC DS1307 module using its functions: **uRTCLib()** (used for the RTC I2C address which is usually 0x68), **URTCLIB_WIRE.begin()** (used for communicating via I2C), **set()** (used for setting RTC parameters), refresh(), dayOfWeek(), day(), month(), year(), hour() and minute().

5.2.3. Key Functionalities

- **Normal Operation** – The loop function continuously polls if **"SET" button** is pressed, then it decides what the MCU should do according to **"SET_BTN" software flag**. Normally, the MCU just updates the temperature on the LCD via the "**readSensorsAndDisplay()**" function. However, if the user has just exited "setting mode", the date and time are also updated. Finally, if neither of these actions are done, the MCU goes (back) to **sleep**.
- **Low-Power Strategies** – The MCU is put to sleep (Power-Down Mode) using an inline assembly instruction. It periodically wakes-up (for very little time, just a few machine cycles) by the WatchDog Timer, which is set to 8s intervals (largest interval available for the MCU). However, with the external interrupt being active, it can still wake up the MCU from the Power Down mode and the Interrupt Service Routine is triggered.

- **Setting Mode (FSM implementation)** – The setting mode logic is implemented using A **Moore-like Finite State Machine** of which implementation **diagram** can be observed in the figure below. The main advantage of a Moore FSM for this project is its robustness and low implementation complexity.



FSM Diagram

5.2.4. Example Code Snippets

In the following part I will present you some of the most **relevant code snippets** along with relevant comments.

One important function is the one which disables the unused pins of the MCU:

```
 void disableUnusedPins(){
   const byte usedPins[] = {rs, en, d4, d5, d6, d7, DHT_PIN, A4, A5, 2}; //defining the pins which should not be disabled
   bool isUsed;                                        //flag for used pins

   for(byte i=0; i<20; i++){                           //for loop that iterates through all Arduino Pro Mini pins and flags them
     isUsed = false;
     for (byte j=0; j<sizeof(usedPins); j++){          //for loop that compares the previously defined pins and saves them from
                                                       //being disabled
       if (i == usedPins[j]){
         isUsed = true;
         break;
       }
     }
     if (!isUsed){
     pinMode(i, OUTPUT);
```

Another important function is the one which updates the display with new readings form the DHT-11 sensor and also updates the time and date based on the readings from the DS1307:

```
void readSensorsAndDisplay(){         //function used for making sensor reads and updating the LCD
  rtc.refresh();

//displaying the Day Of The Week, month/day and time
  lcd.begin(16,2);
  lcd.clear();
  lcd.setCursor(1,0);
  lcd.print(daysOfTheWeek[rtc.dayOfWeek()-1]);
  lcd.print(" ");

//formatting date: MM/DD
  if (rtc.month() < 10){
    lcd.print('0');
  }
  lcd.print(rtc.month());lcd.print('/');

  if (rtc.day() < 10){
    lcd.print('0');
  }
  lcd.print(rtc.day());

  lcd.print(" ");
```

```
//formatting time: hh:mm
  if (rtc.hour() < 10){
    lcd.print('0');
  }
  lcd.print(rtc.hour());lcd.print(':');
  if (rtc.minute() < 10){
    lcd.print('0');
  }
  lcd.print(rtc.minute());

//printing temperature on the second row of the LCD
  lcd.setCursor(2,1);
  lcd.print("Temp: ");
  if (DHT.temperature < 10) {
    lcd.print('0');
  }
  lcd.print(DHT.temperature);
  lcd.print((char)223);           /*"°" symbol*/
  lcd.print("C");
 }
```

Two more crucial functions are those which disable and enable Watchdog Timer and INT0 interrupts so that when in setting mode the MCU can't be interrupted as it would normally do:

```
/* --function that disables Watchdog and INT0 interrupts (called when entering setting mode)-- */
void disableWDTandINT0(){
  detachInterrupt(0);
  cli();
  WDTCSR |= (1 << WDCE) | (1 << WDE);
  WDTCSR = 0x00;
  sei();
}

/* --function that enables Watchdog and INT0 interrupts (called when exiting setting mode)-- */
void enableWDTandINT0(){
  attachInterrupt(0, digitalInterrupt, RISING);
  cli();
  WDTCSR = (24);              //sets Watchdog into Interrupt Mode
  WDTCSR = (33);              //sets Watchdog Timer at 8s Time-outs
  WDTCSR |= (1<<6);           //enables watchdog interrupts
  sei();
}
```

Finally, the function that controls the setting mode in a FSM logic is very important for this project:

```
void enterSettingMode(){

  disableWDTandINT0();        //disabling WDT and INT0 interrupts in order to make sure that all setting steps are set

  rtc.refresh();              //using local variables for RTC parameters (keeps code tidy)
  byte day = rtc.day();
  byte month = rtc.month();
  byte year = rtc.year();
  byte dow = rtc.dayOfWeek();
  byte hour = rtc.hour();
  byte minute = rtc.minute();

  bool set = true;            //flag used for setting mode
  byte settingStep = 0;       //defining the FSM state

  lcd.clear();

  while(set){
    lcd.setCursor(0,0);
    lcd.print("             ");
    lcd.setCursor(0,0);

/*
 * FSM controlled by "settingStep" variable, which is modified by SET button.
 * Each step displays different text based on the current state.
 * The data is also updated in real time as the user presses INC/DEC buttons.
*/
  switch(settingStep){
    delay(50);
      case 0: lcd.print("Set Day: "); lcd.print(day); break;
      case 1: lcd.print("Set Month: "); lcd.print(month); break;
      case 2: lcd.print("Set Year: "); lcd.print(year); break;
      case 3: lcd.print("Set DOW: "); lcd.print(daysOfTheWeek[dow - 1]); break;
      case 4: lcd.print("Set Hour: "); lcd.print(hour); break;
      case 5: lcd.print("Set Minute: "); lcd.print(minute); break;
  }

  if(digitalRead(SET_PIN) == HIGH){                       //function that controls FSM state by polling for the SET button
    delay(200);
    settingStep++;
    if(settingStep > 5){
      rtc.set(10, minute, hour, dow, day, month, year);     //after setting everything up, the RTC module is updated with
                                                            //the new data
    lcd.clear();
    SET_BTN = false;                                        //making sure the SET button is not stuck on "true"
    while(digitalRead(SET_PIN) == HIGH){                    //debouncing the SET button
      delay(50);
    }
    set = false;
  }
  }
```

```
if(digitalRead(INC_PIN) == HIGH){                        //polling for the INC button and updating data corresponding to
                                                         //current FSM state
    delay(100);
    switch(settingStep){
        case 0: if(day < 31) day++; break;
        case 1: if(month < 12) month++; break;
        case 2: if(year < 99) year++; break;
        case 3: if(dow < 7) dow++; else dow = 1; break;
        case 4: if(hour < 23) hour++; break;
        case 5: if(minute < 59) minute++; break;
    }
  }

  if(digitalRead(DEC_PIN) == HIGH){                        //polling for the DEC button and updating data corresponding to
                                                           //current FSM state
    delay(100);
    switch(settingStep){
        case 0: if(day > 1) day--; break;
        case 1: if(month > 1) month--; break;
        case 2: if(year > 24) year--; break;
        case 3: if(dow > 1) dow--; else dow = 7; break;
        case 4: if(hour > 0) hour--; break;
        case 5: if(minute > 0) minute--; break;
    }
  }

  delay(50);
  }

  lcd.clear();
  SET_BTN = false;                                         //making sure the SET button is not stuck on "true"
  enableWDTandINT0();                                      //enabling Watchdog Timer and INT0 interrupts
}
```

6. Testing and Validation

6.1. Test Plan

The testing is a very important phase for developing a prototype so during this phase I have actively **tested the features** I was implementing, one by one and finally the device as a whole system.

- One feature that was easy to test was if the **display output** is clean (readable and user-friendly), not glitching and displaying the right data.
- The **DS1307 RTC** Module also required some testing to make sure that it keeps working and the time and date are updated in real time even when the **main supply** (9V battery) is **disconnected**. Also it required some **I2C bus testing** to make sure that the **communication** between the MCU and the module is done correctly and the **integrity of the data** is kept.
- The **buttons**, which make up the **UI**, also had to be tested tremendously tested so that the user does not encounter any bugs when interacting with the device.
- Finally, the key aspect of this device is that it runs on **batteries**, so the supply energy is limited and tests were done in different consumption scenarios in order to achieve the **optimum power efficiency**.

6.2. Test Results

- At some point after I have integrated the DS1307 module in the project and started to **display data from it** on the LCD display there were some problems with the **date and time** that were displayed. Sometimes an **incorrect date and time** would be displayed (e.g. 0/0 00:00) or instead of the normal date and time format there would be **random ASCII characters** displayed.
- After testing the LCD output and seeing that the date displayed is not correct, the next probable cause would be the **RTC module not communicating correctly** with the MCU. With this module there could be two problems: the module would not work without the **main power supply** (the circuit that selects either the 5V line or the VBAT as the power supply for the DS1307 is not working) or the **I2C messages are not received correctly** from the module.
- When testing the **buttons** the most important thing to keep in mind was the **debouncing**. Sometimes the "**SET" button** would glitch and the MCU would poll for it too fast and count **one press as two separate inputs**, skipping setting steps or glitching in setting mode loop right after last step.
- The last feature implemented was the autonomous feature, the 9V battery as the main power supply for the device. The PP3 9V battery has around 500mAh which is not that much considering that the device average measured power consumption is around 105 mA.

6.3. Problems and Solutions

- After acknowledging the **incorrect data format displaying problem** on the LCD I've started to dive deeper into the problem and analyze it on a multi-level perspective. At first, I've checked all the **hardware connections** that could be **loose or broken**. After making sure all the connections were good, I have moved on to the next probable cause which could be the **LCD display code**. After reassuring that the LCD updates every time needed the last thing left to check was the **DS1307 RTC module**.
- Moving to the **DS1307 module**, the first thing to check was the **I²C bus**. In order to make sure that the module and the MCU are communicating right via I²C I've used a **small code snippet** that outputted in the serial monitor if a device is connected on the I²C bus and if so, **its address**:

  At first, the code output would be "**No I2C devices found**", which was unusual because at the **eye-level** all the **connections seemed fine** and a device was supposed to be found at address **0x68** (specific address for DS1307 IC) so I had to overlook all the electronic traces from the MCU to the RTC module again. After not being able to observe any problems I've decided to **resolder** all the connections and after that the module was **found at address 0x68**, meaning that the problem was **a hardware one**.

```
#include <Wire.h>

void setup() {
  Wire.begin();
  Serial.begin(9600);
  while (!Serial);
  Serial.println("\nI2C Scanner");
}

void loop() {
  byte error, address;
  int nDevices = 0;

  Serial.println("Scanning...");

  for (address = 1; address < 127; address++) {
    Wire.beginTransmission(address);
    error = Wire.endTransmission();

    if (error == 0) {
      Serial.print("I2C device found at address 0x");
      if (address < 16) Serial.print("0");
      Serial.print(address, HEX);
      Serial.println("  !");

      nDevices++;
    }
  }

  if (nDevices == 0) {
    Serial.println("No I2C devices found\n");
  } else {
    Serial.println("Done.\n");
  }

  delay(2000);
}
```

- After solving the problem with the I²C bus communication I could move on to solving the **debouncing problem** with the buttons. At first, the problem was that when the **"SET" button** was pressed when the MCU would happen to be in the power-down mode, after all the setting steps were done and "SET" button was **pressed one last time**, it would **not return into normal display**, instead **no information would be displayed** until some time passed. This problem was due to the fact that when the MCU would **enter setting mode from the power-down mode** (waking it up) when it returned to the main code it started compiling from the last line, which was the power-down routine where it would **wait for the "sleep"**

loop to finish then run the loop code again and **call the function** which **updates the display**. **The solution** to this problem was very simple and clever: I just **added a line of code** where the “readSensorsAndDisplay()” function would be called inside the “sleep” routine:

```
for(byte i=0; i<5; i++){
    if(!SET_BTN)
       __asm__  __volatile__("sleep");
    else{
      SET_BTN = false;
      delay(100);
      enterSettingMode();
      ?????? missing line that caused glitching ??????
    while (digitalRead(SET_PIN) == HIGH){
      delay(50);
    }
```

```
for(byte i=0; i<5; i++){
    if(!SET_BTN)
       __asm__  __volatile__("sleep");
    else{
      SET_BTN = false;
      delay(100);
      enterSettingMode();
      readSensorsAndDisplay();
    while (digitalRead(SET_PIN) == HIGH){
      delay(50);
    }
```

However, this was **not the only issue** with the buttons. Sometimes if the “SET” button was **pressed for a longer time** the MCU would count that as **two or more fast consecutive inputs** and would **jump over setting steps**. To solve this issue **some delays were added** after each setting step was modified. The **“SET_BTN” flag** was also **set to false** after each time it could be modified in order to make sure that the interrupt does not occur randomly.

7. Conclusions

7.1. Project Achievements

There are many things I've achieved developing this device, but the most important for me is that I have designed from scratch a fully functional and autonomous device that is very portable and can be easily reproduced by anyone who follows this documentation. Another important achievement would be that during the course of developing this device I've gained a new perspective on autonomous devices and that is the power efficiency. This project made me come up with solutions when it comes working with limited power supplies (in my case a PP3 battery). Finally, due to the bugs and problems encountered along the way I was able to improve my debugging skills, given that I've worked with some modules I haven't used before.

7.2. Lessons Learned

As I previously mentioned about the new perspective unlocked, I've learned how to integrate inline assembly in my code and also work with system interrupts and external interrupts (using flags), since the device functions most of the time in a "power-down" mode in which the interactions with the device are limited.

Working with batteries also helped me improve when it comes to choosing the modules used in certain projects, especially ones that have power constraints. It is very important to use the appropriate modules that are just enough for the needs of the project.

Something that was new to me was working with Autodesk Fusion and designing the custom enclosure for the device. I've learned how to design basic shapes and how to work with multiple different pieces that fit together. Also Autodesk Fusion comes with a circuit and PCB design tool, which of whom I've used the first one for designing the schematic.

The FSM I have implemented in C++ was also something new to me. The choice of a Moore FSM was due its robustness and ease of implementation; also this type of FSM is rather more intuitive than a Mealy FSM.

7.3. Future Improvements

To conclude, here are some future improvements that can be implemented keeping the main core of the project intact:

Low-Power Optimization: The first thing I'd improve is the power supply in order to extend the device's battery life and portability. The PP3 battery can be replaced with 4 AA batteries which would have around 2500mAh (5 times greater than current battery) or for even more portability and battery life, 2 18650 Lithium-Ion batteries (require charging circuitry) that can be recharged from a solar cell or a common USB-C cable. Both of these options regarding the power supply would also benefit if a step-down converter circuit would be added in order to maximize battery life.

Battery Management System: This upgrade would be mandatory if a power cell is added, however the BMS can be implemented either way as it would help to collect data about battery life in real time and display it so that the user can predict when the battery needs replacement or charging (for 18650 Li-Ion cells).

Upgrading to an OLED display: An OLED display has a higher resolution and also is very power-efficient. This type of display would minimize the overall power consumption (a small OLED display would need no more than 30mA compared to 100mA used for the 1602 LCD) of the device and would also give it a cleaner aspect.

Wireless or BLE connection: A very nice to have feature is to be able to connect to the device using either one of these connection and get real-time data on the phone. However this upgrade would require some extra modules that need to be adapted to the device's main constraints which is power consumption and small size. Also, being able to connect with a phone would allow the time to be set and synchronized with a phone, eliminating manual setting.

Auto-Dimming LCD: A light sensor (LDR) can be added to sense the ambient lightning level and make automatic adjustments to the LCD backlight, improving the device power consumption and making it more autonomous.

PCB Design: Having the circuit already implemented in Autodesk Fusion would make it even easier to design a PCB that would make the device even more stable (removing potential EMC problems) and even help designing a smaller enclosure.

8. References

Arduino Pro-Mini General Data - https://docs.arduino.cc/retired/boards/arduino-pro-mini/
1602 LCD General Data - http://wiki.sunfounder.cc/index.php?title=LCD1602_Module
- https://docs.arduino.cc/learn/electronics/lcd-displays/
DS1307 Module General Data - https://wiki.geeetech.com/index.php/Real_Time_Clock_RTC_DS1307_Module
I2C Protocol - https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

Tutorials

I2C Tutorial - https://www.youtube.com/watch?v=CAvawEcxoPU&t=288s
Arduino Sleep Tutorial - https://www.youtube.com/watch?v=urLSDi7SD8M&t=1138s

Libraries

LCD Library - https://docs.arduino.cc/libraries/liquidcrystal/
DHT11 Library - https://github.com/RobTillaart/DHTlib?tab=readme-ov-file
RTC Library - https://github.com/Naguissa/uRTCLib?tab=LGPL-3.0-1-ov-file

Official Datasheets

ATMega328P
https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf
DS1307
https://www.e-gizmo.net/oc/kits%20documents/TinyRTC%20I2C%20module/TinyRTC%20i2c%20module%20%20Techincal%20Manual%20rev1.pdf
DHT11
https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf?srsltid=AfmBOorbLVRi8RRHqe2lIjBTA0UB3BlBdAoubiNu2XRn_OGft9KSbzLm

9. Appendices

The complete source code with relevant comments can be found *here*.
The .stl files that are ready for 3D printing can be found *here*.
The circuit schematic can be found here.