# ALARM CLOCK

## REPORT, GROUP F9

Thursday, 10:15 AM laboratory
Date: 06.12.2018
IFE Information Technology, 5th semester

TEAM LEADER Ewa Rudak (210950)
Matteo Belenchia (902765)
Emilio Silvestri (902770)
Mateusz Wasilewski (210954)

## Contents

## Project description

The project chosen by our group is an alarm clock that can be turned off by playing and completing a very simple game. In order to make the device more effective in serving its purpose, not only did we change the standard method of switching off the alarm into a more complicated one, but also we set a simple alarm melody instead of a one-note one. The reason we did so was our own experience with alarm clocks, which were unable to effectively get us up. Our group wanted to live up to the expectations of those who have problems with falling asleep after disabling their alarms with one click. Moreover, the device displays the temperature and humidity and allows the user to set the current date and time.

## The division of responsibilities

The members of the group were responsible for the following parts of the project:

- **Mateusz Wasilewski** used the LCD Nokia display and joystick to make the simple game for our project;
- **Matteo Belenchia** was responsible for the buttons (black buttons, red button and the DPDT switch), DHT11 thermometer and humidity sensor, and the photoresistor;
- **Emilio Silvestri** took care of the 7-segment display and the RTC clock;
- **Ewa Rudak** was responsible for the buzzer and the 8x8 LED matrix.

## User guide

The device can be turned on by simply connecting it to the power source, and turned off by disconnecting it. (Figure 1)

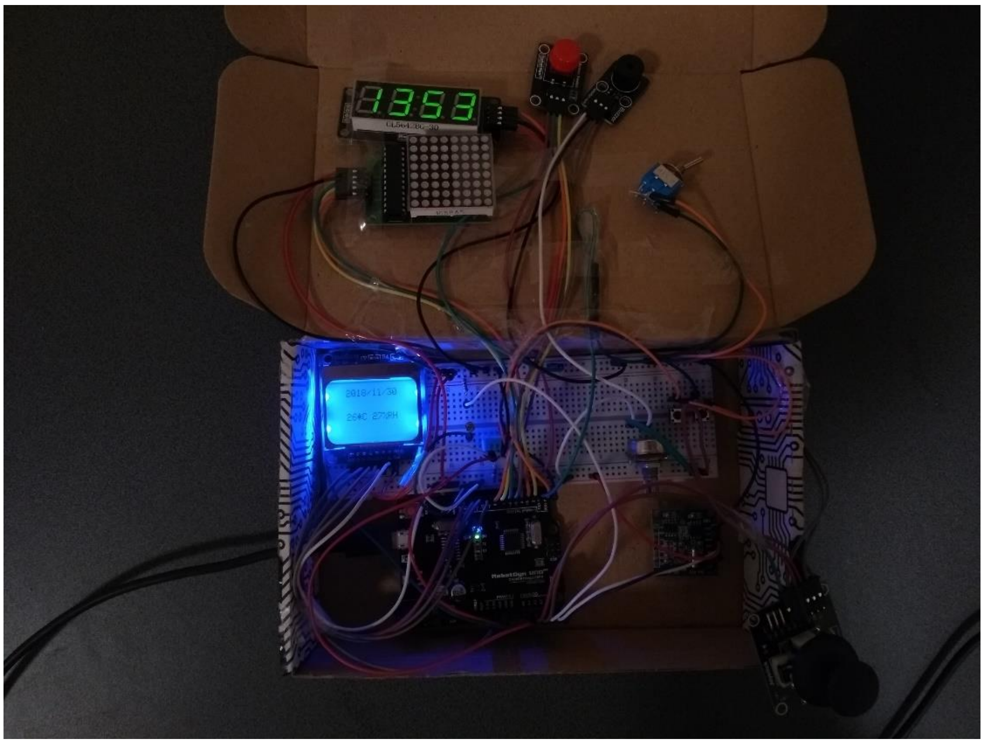


*Figure 1*

The rightmost black button allows you to switch the display modes between the alarm and clock. When using the clock for the first time, you should set the current date and time.(Figure 2, Figure 3)

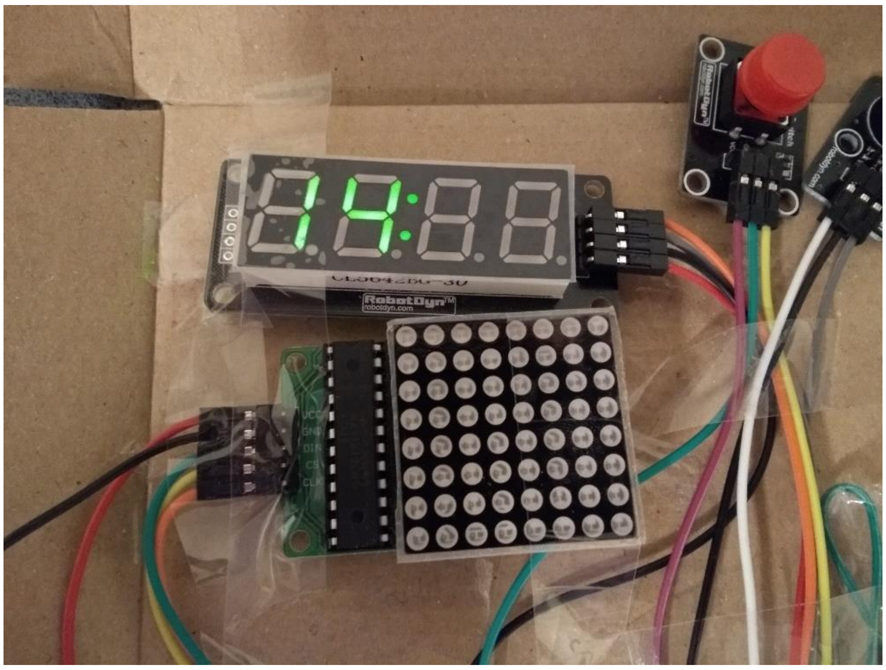


*Figure 2*

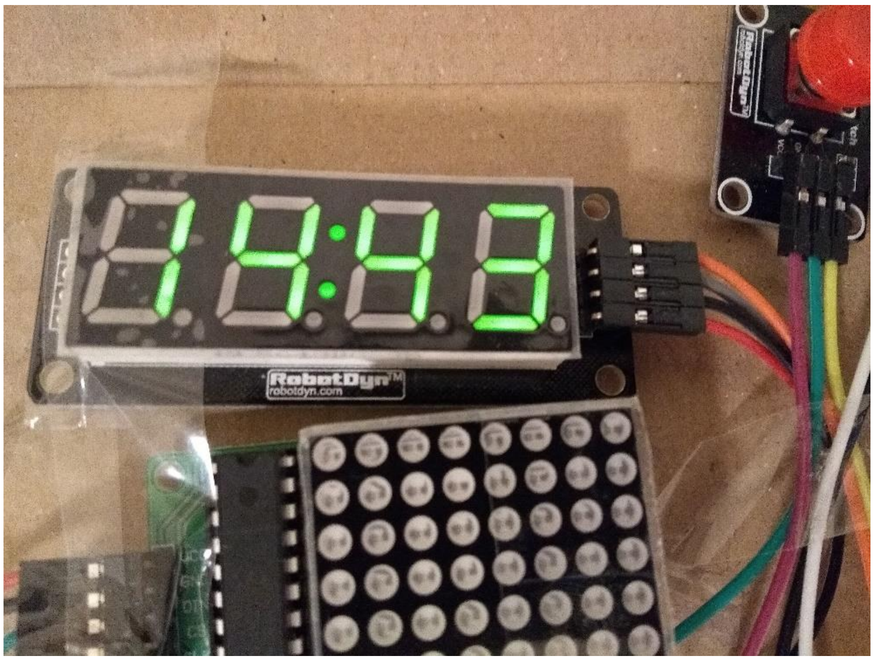


*Figure 3*

You can do it by simply long pressing the red button. The next step is setting the hours, minutes, years, months, and days using the black buttons to increase/decrease the number and confirming the changes by short pressing the red button each time until all the values have been set. (Figure 4, Figure 5, Figure 6)

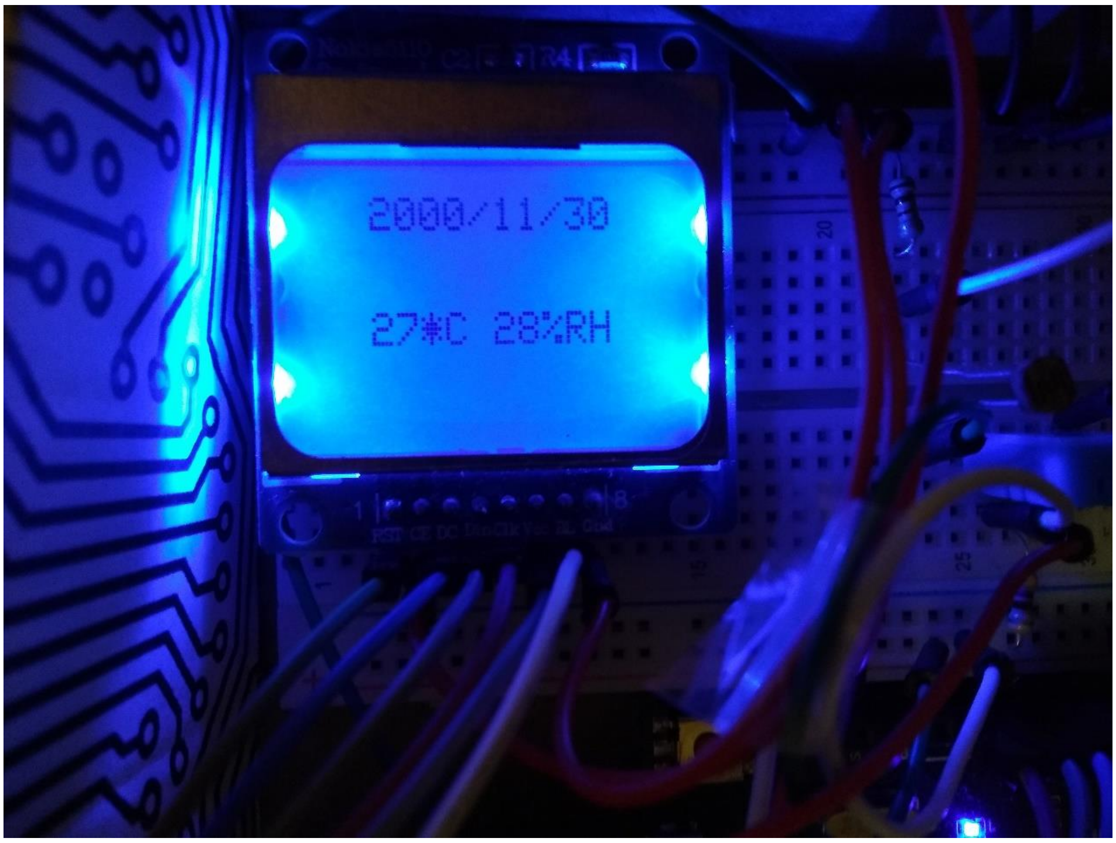


*Figure 4*

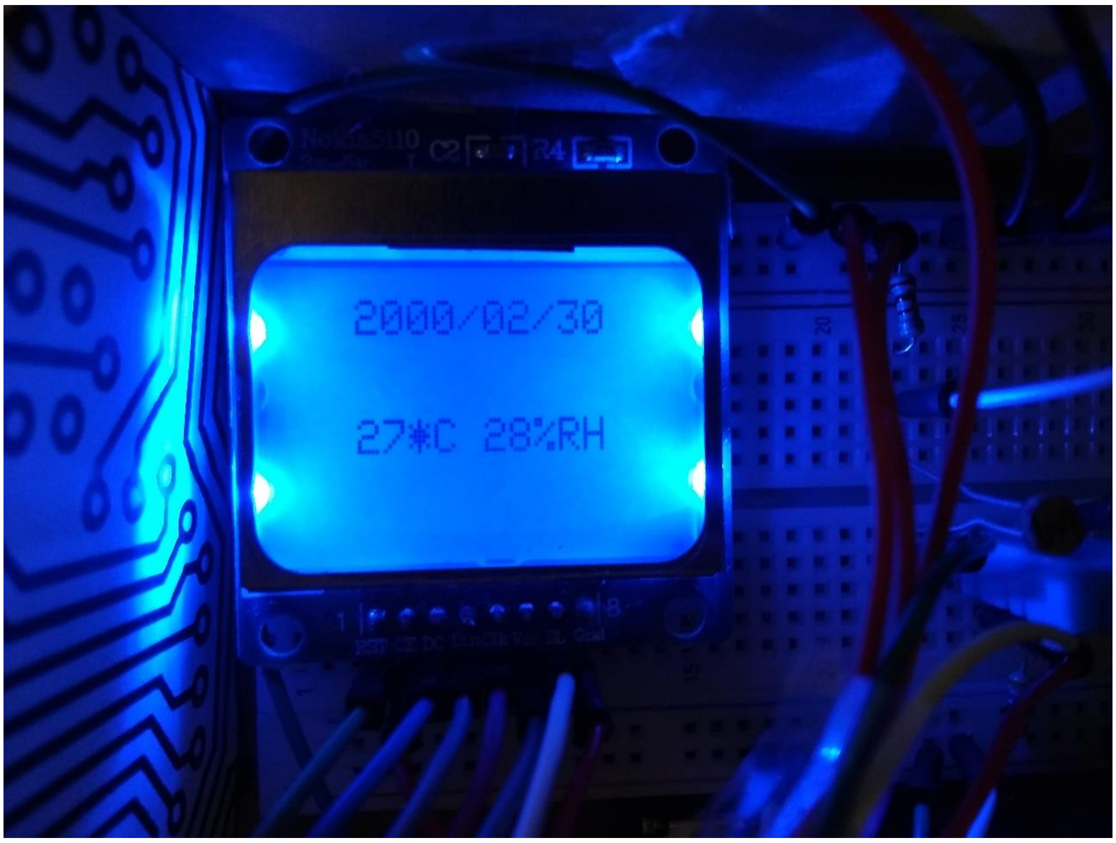


*Figure 5*

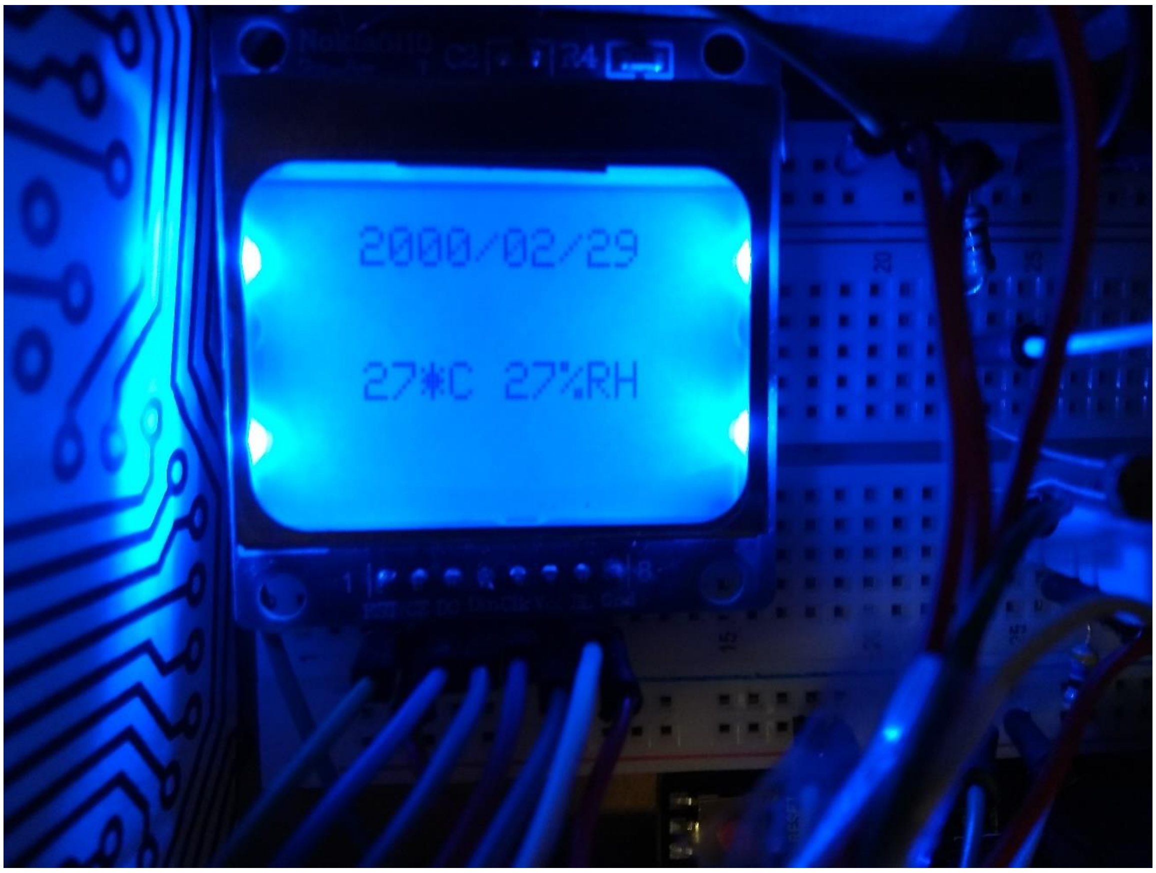


*Figure 6*

Setting the alarm is equally simple – first you should short press the red button and then keep pressing the black buttons until the alarm time has been set.
The black button on the right is used to increase the values, the one on the left decreases them.
In order to enable the alarm, the switch button should be set to LOW. (Figure 7)

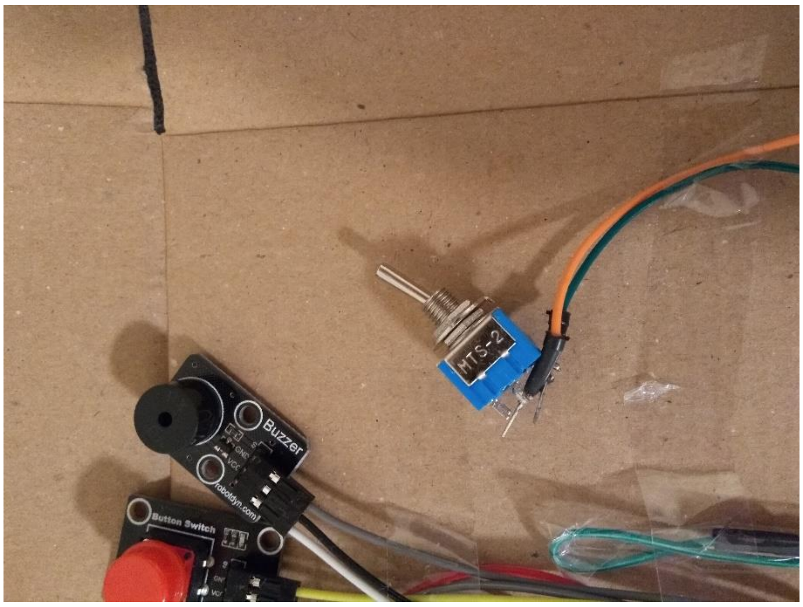


*Figure 7*

The 7-segment display shows the current or alarm time, depending on the chosen display mode. The date, humidity and temperature are displayed on the LCD Nokia display. It is also used for the game.
After the state changes, you can see the animations on the 8x8 display. (Figure 8, Figure 9)

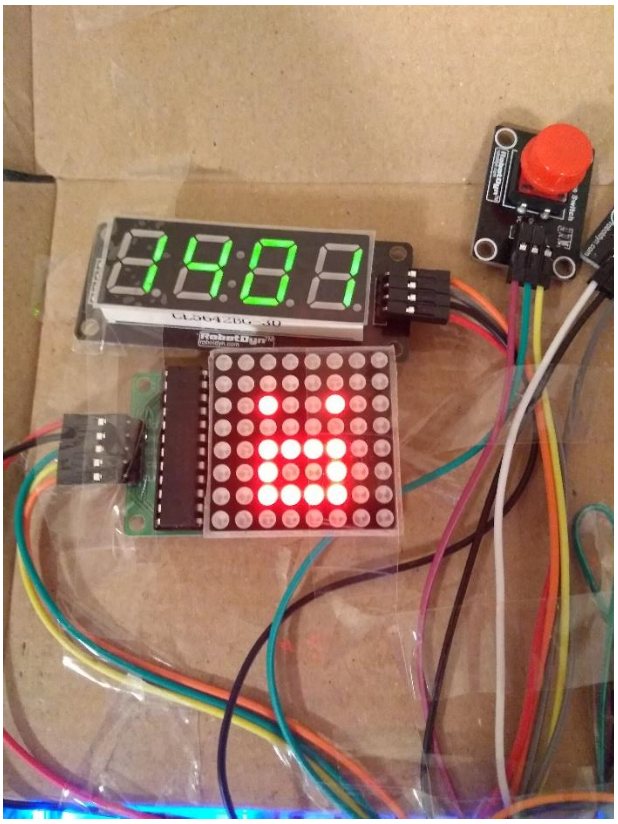


*Figure 8*

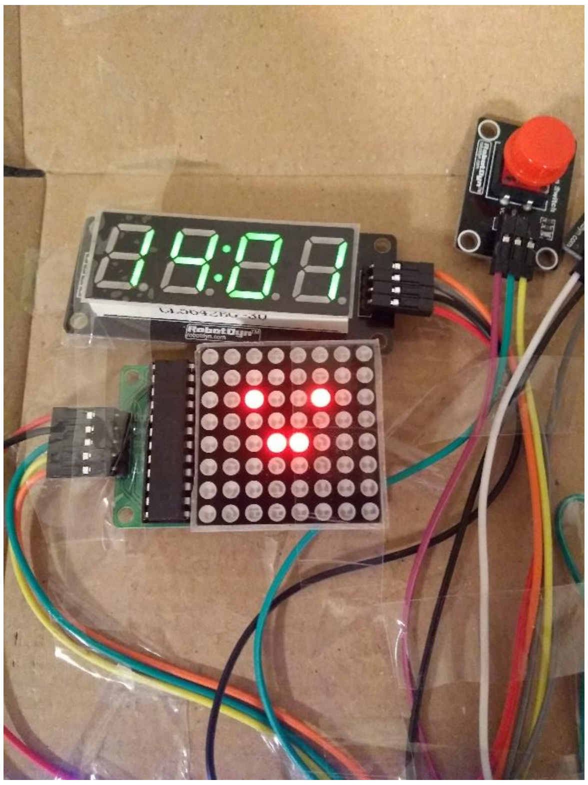


*Figure 9*

If you want to increase or decrease the volume of the buzzer, you can use the potentiometer. Turn it to set the desired volume.

The photoresistor is used to adjust the brightness of the 7-segment display and the 8x8 LED matrix. The brightness of the displays is enhanced when the decrease of the brightness of the environment is noticed by the photo-conductive cell.

In order to turn off the alarm clock when it is buzzing, you have to toggle the switch to the HIGH position and play the game displayed on the LCD display. To win it, you have to repeat the pattern of arrows displayed using the joystick.  The time for each move is set to 200 ms. Once the game is completed and the switch set to HIGH, the buzzer stops ringing.(Figure 10, Figure 11)

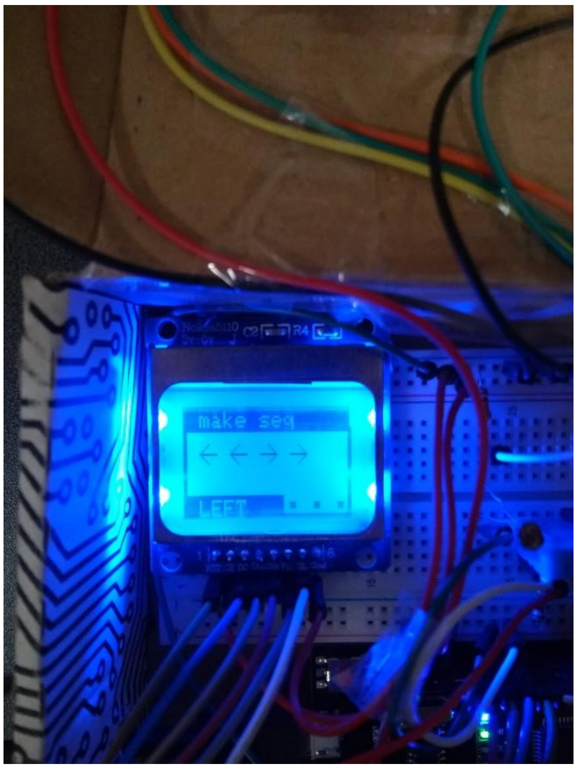


*Figure 10*

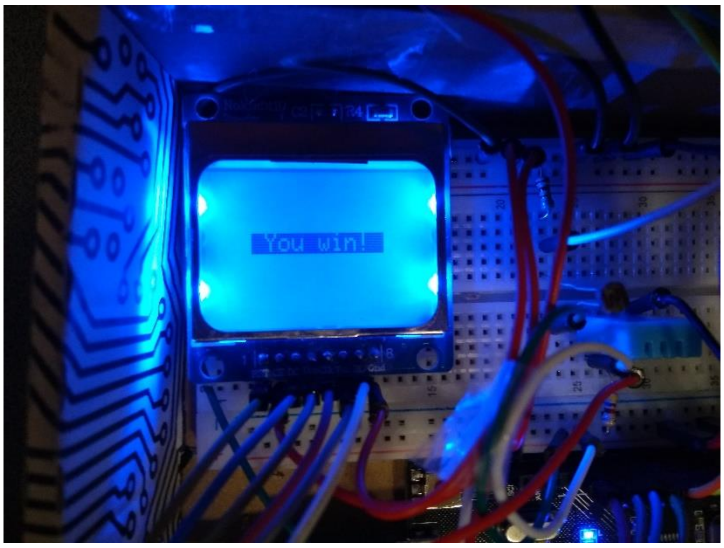


*Figure 11*

## Arduino UNO

Arduino Uno is a microcontroller board based on the ATmega328P MCU. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz quartz crystal, a USB connection, a power jack, an ICSP header and a reset button.[1]

We are not using an original Arduino board, but given the open source nature of the Arduino project, we got a completely compatible board with some slight differences, namely it features a micro USB connector instead of a USB one, the CH340G chip for UART and has 2 extra analog only pins which we won't use to make our project Arduino Uno compatible.[2]

### Power

The Arduino UNO can be powered via the micro USB connection or with an external power supply. The power source is selected automatically.

The power pins are as follows:

- VIN: The input voltage to the Arduino board when it's using an external power source (as opposed to 5 volts from the USB connection or other regulated power source). Voltage can be supplied through this pin, or, if supplying voltage via the power jack, it can be accessed through this pin.
- 5V: The regulated power supply used to power the microcontroller and other components on the board. The board can be supplied with power either from the DC power jack (7 - 12V), the USB connector (5V), or the VIN pin of the board (7-12V). Supplying voltage via the 5V or 3.3V pins bypasses the regulator, and can damage your board.
- 3V3: A 3.3 Volt supply generated by the on-board regulator. Maximum current draw is 50 mA.
- GND: Ground pins.
- IOREF: The voltage at which the i/o pins of the board are operating (i.e. VCC for the board).

### Memory

The ATmega328 has 32 KB (with 0.5 KB occupied by the bootloader). It also has 2 KB of SRAM and 1 KB of EEPROM (which can be read and written with the EEPROM library).

### Input and Output

Each of the 14 digital pins on the Uno can be used as an input or output, using pinMode(),digitalWrite(), and digitalRead() functions. They operate at 5 volts. Each pin can provide or receive 20 mA as recommended operating condition and has an internal pull-up resistor (disconnected by default) of 20-50k ohm. A maximum of 40mA is the value that must not be exceeded on any I/O pin to avoid permanent damage to the microcontroller.
In addition, some pins have specialized functions:

- Serial: 0 (RX) and 1 (TX). Used to receive (RX) and transmit (TX) TTL serial data. These pins are connected to the corresponding pins of the ATmega8U2 USB-to-TTL Serial chip.
- TWI (I2C equivalent): A4 or SDA pin and A5 or SCL pin. Support TWI communication using the Wire library.
- External Interrupts: 2 and 3. These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value.
- PWM: 3, 5, 6, 9, 10, and 11. Provide 8-bit PWM output with the analogWrite() function.

[1] https://store.arduino.cc/arduino-uno-rev3
[2] https://robotdyn.com/uno-r3-smt-atmega328-usb-serial-ch340g-micro-usb.html

- SPI: 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). These pins support SPI communication using the SPI library.
- LED: 13. There is a built-in LED connected to digital pin 13. When the pin is HIGH value, the LED is on, when the pin is LOW, it's off.
- Analog Inputs: The Uno has 6 analog inputs, labeled A0 through A5, each of which provide 10 bits of resolution (i.e. 1024 different values). By default they measure from ground to 5 volts, though is it possible to change the upper end of their range using the AREF pin and the analogReference() function.
- AREF: Reference voltage for the analog inputs. Used with analogReference().
- Reset: Bring this line LOW to reset the microcontroller. Typically used to add a reset button to shields which block the one on the board.

## Communication

Arduino Uno has a number of facilities for communicating with a computer, another Arduino board, or other microcontrollers. The ATmega328 provides UART TTL (5V) serial communication, which is available on digital pins 0 (RX) and 1 (TX). An ATmega16U2 on the board channels this serial communication over USB and appears as a virtual com port to software on the computer. The 16U2 firmware uses the standard USB COM drivers, and no external driver is needed. However, on Windows, a .inf file is required. The Arduino Software (IDE) includes a serial monitor which allows simple textual data to be sent to and from the board. The RX and TX LEDs on the board will flash when data is being transmitted via the USB-to-serial chip and USB connection to the computer (but not for serial communication on pins 0 and 1).A SoftwareSerial library allows for serial communication on any of the Uno's digital pins. The ATmega328 also supports I2C (TWI) and SPI communication. The Arduino software includes a Wire library to simplify use of the I2C bus.

## Programming

The Arduino Uno can be programmed with the Arduino Software IDE. The ATmega328 on the Arduino Uno comes preprogrammed with a bootloader that allows you to upload new code to it without the use of an external hardware programmer. It communicates using the original STK500 protocol. You can also bypass the bootloader and program the microcontroller through the ICSP (In-Circuit Serial Programming) header using Arduino ISP or similar.

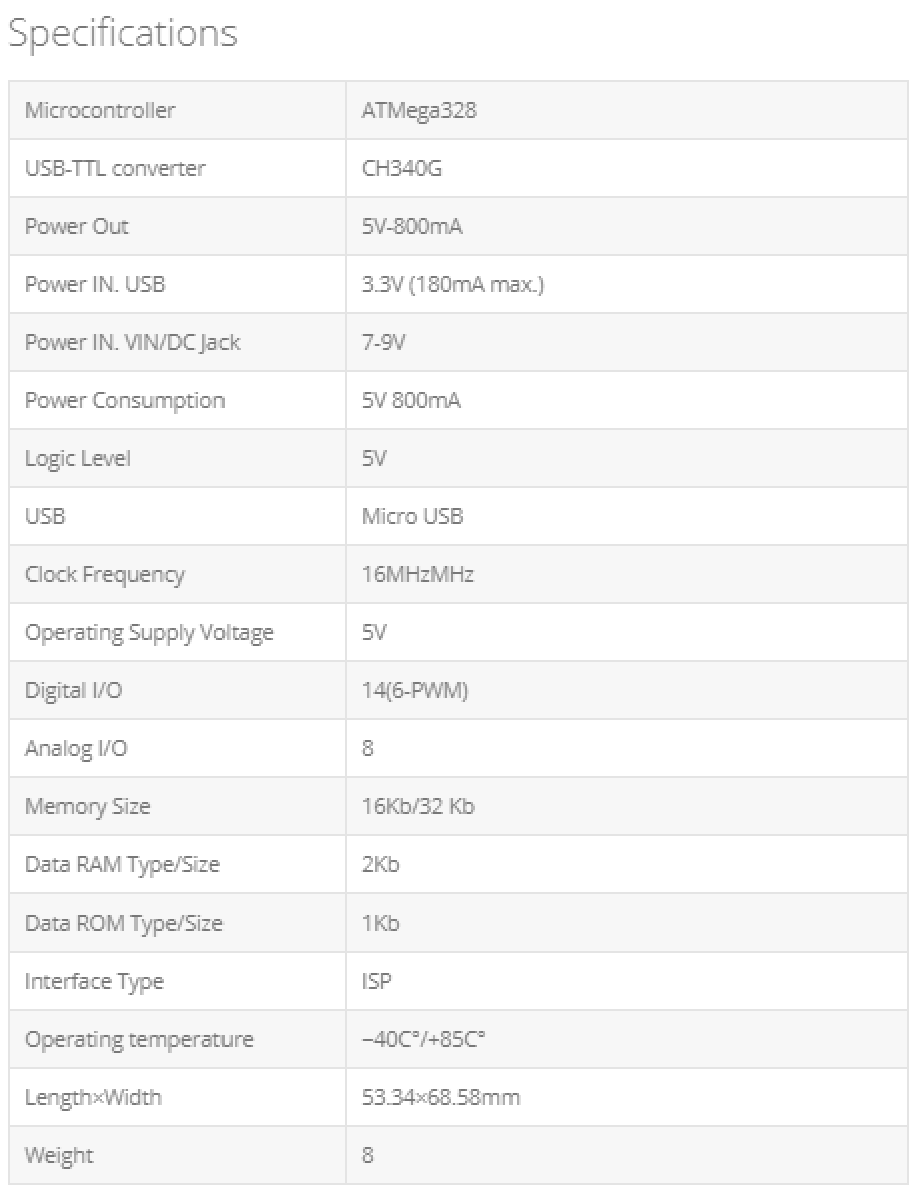
## Automatic (Software) Reset and Bootloader Initiation

Rather than requiring a physical press of the reset button before an upload, the Arduino Uno is designed in a way that allows it to be reset by software running on a connected computer. One of the hardware flow control lines (DTR) of the ATmega8U2/16U2 is connected to the reset line of the ATmega328 via a 100 nanofarad capacitor. When this line is asserted (taken low), the reset line drops long enough to reset the chip. The Arduino Software (IDE) uses this capability to allow you to upload code by simply pressing the upload button in the interface toolbar. This means that the bootloader can have a shorter timeout, as the lowering of DTR can be well-coordinated with the start of the upload. This setup has other implications. When the Uno is connected to either a computer running Mac OS X or Linux, it resets each time a connection is made to it from software (via USB). For the following half-second or so, the bootloader is running on the Uno. While it is programmed to ignore malformed data (i.e. anything besides an upload of new code), it will intercept the first few bytes of data sent to the board after a connection is opened. If a sketch running on the board receives one-time configuration or other data when it first starts, make sure that the software with which it communicates waits a second after opening the connection and before sending this data. The Uno board contains a trace that can be cut to disable the auto-reset. The pads on either side of the trace can be soldered together to re-enable it. It's labeled "RESET-EN". You may also be able to disable the auto-reset by connecting a 110 ohm resistor from 5V to the reset line.

## USB Overcurrent Protection

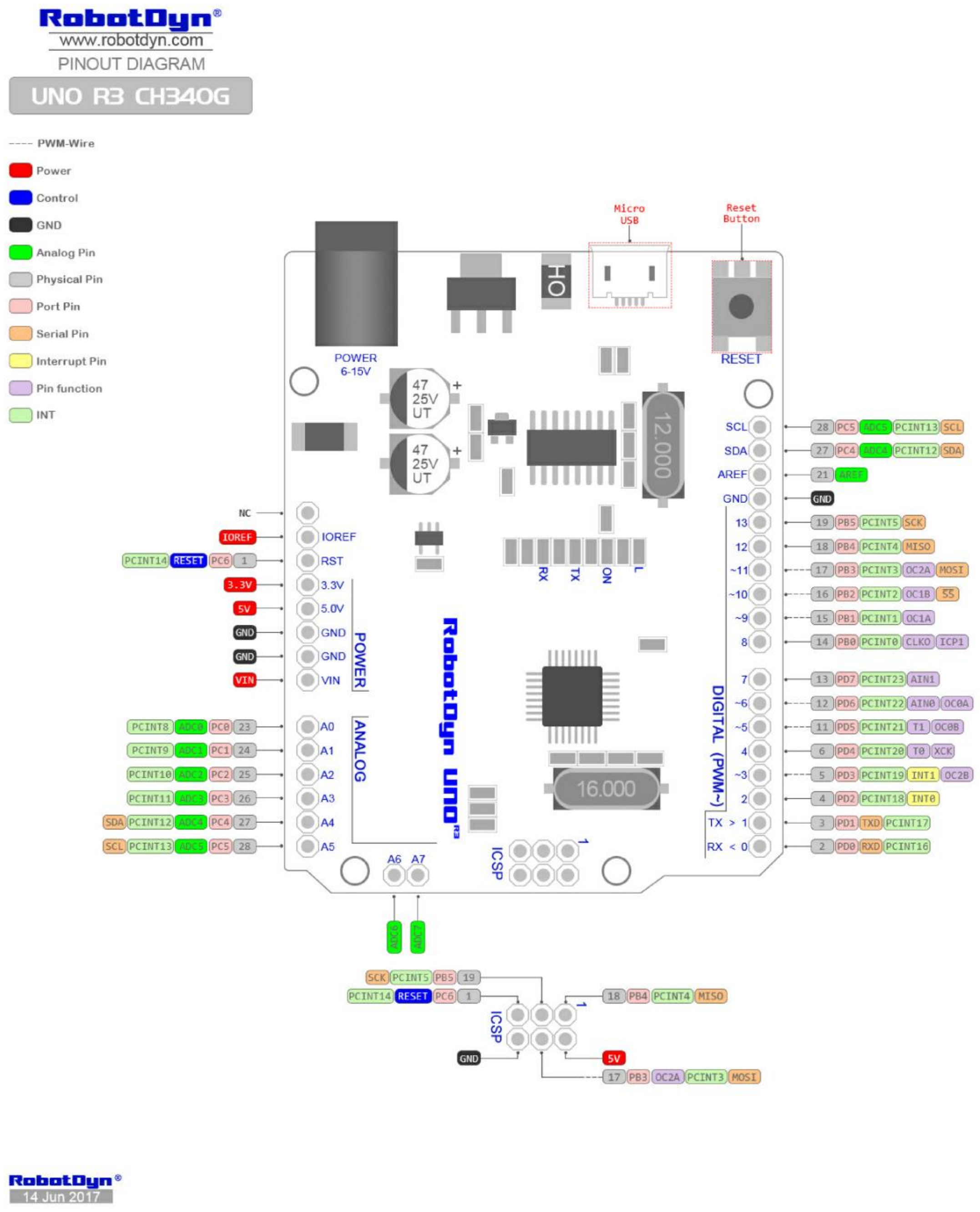
The Leonardo has a resettable polyfuse that protects your computer's USB ports from shorts and overcurrent. Although most computers provide their own internal protection, the fuse provides an extra layer of protection. If more than 500 mA is applied to the USB port, the fuse will automatically break the connection until the short or overload is removed.

## Technical specs



Specifications

| | |
|---|---|
| Microcontroller | ATMega328 |
| USB-TTL converter | CH340G |
| Power Out | 5V-800mA |
| Power IN. USB | 3.3V (180mA max.) |
| Power IN. VIN/DC Jack | 7-9V |
| Power Consumption | 5V 800mA |
| Logic Level | 5V |
| USB | Micro USB |
| Clock Frequency | 16MHzMHz |
| Operating Supply Voltage | 5V |
| Digital I/O | 14(6-PWM) |
| Analog I/O | 8 |
| Memory Size | 16Kb/32 Kb |
| Data RAM Type/Size | 2Kb |
| Data ROM Type/Size | 1Kb |
| Interface Type | ISP |
| Operating temperature | −40C°/+85C° |
| Length×Width | 53.34×68.58mm |
| Weight | 8 |

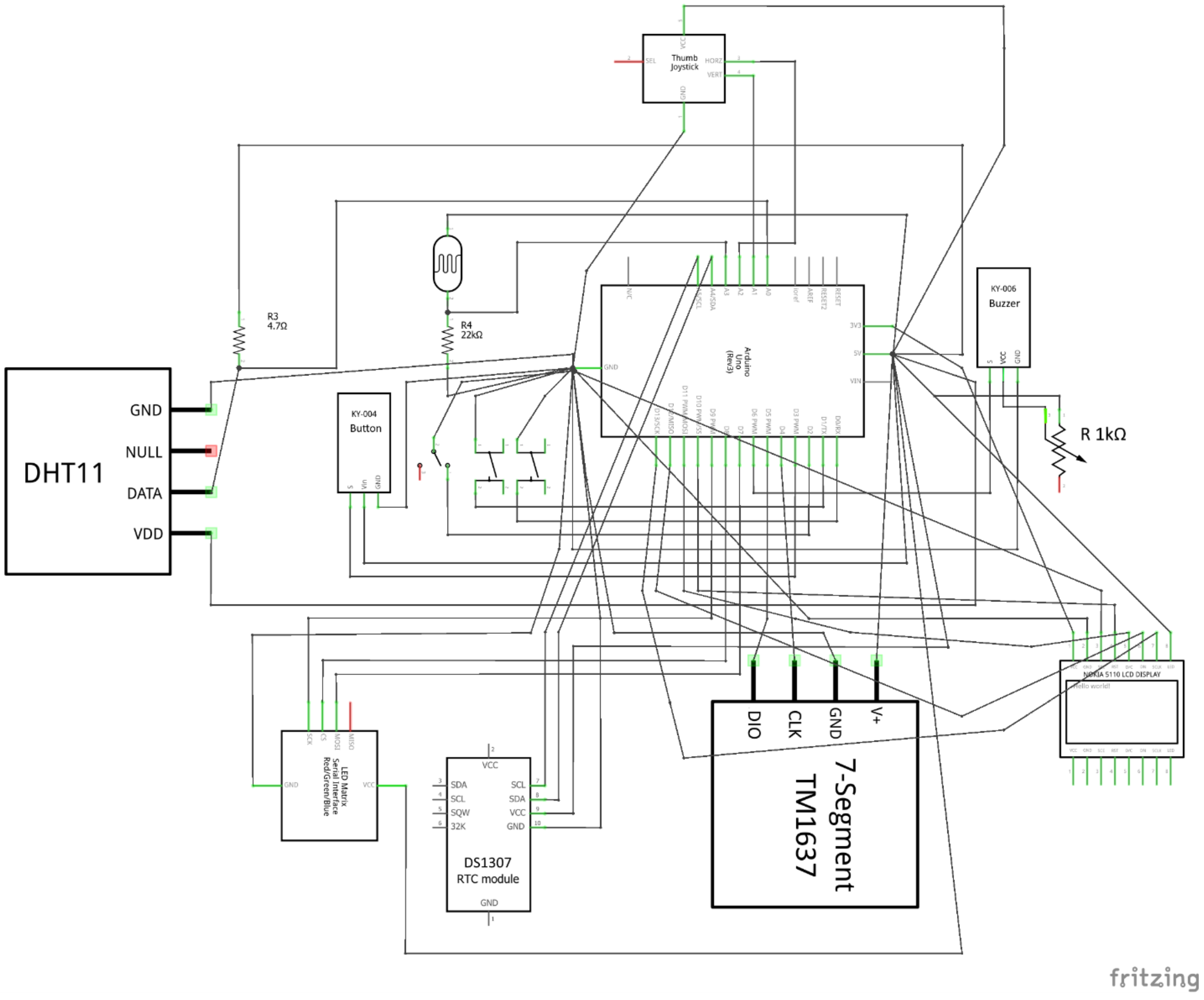


## The components

In the following sections we give a description of the incorporated components, together with a brief hardware specification, a description of the code that manages them, and guidelines about their operation on an MCU register level. Arduino programs commonly make a great use of libraries so programmers know very little about what goes under the hood, unless they want maximum efficiency. Despite the fact that we only ever used libraries in the project we nevertheless added the port manipulation required for each component to work, or the port manipulations required to start and operate the communication protocols. Sometimes these protocols were implemented directly by the library (e.g. software SPI) instead of being embedded in the MCU (hardware SPI). In these cases we reported the relevant functions and wrote the register manipulation details as comments to each line. In order to make our code more understandable, we implied that the label of a bit inside a register has been redefined to its corresponding position, i.e. PORTB

|= (1<<PORTB5) means that 1 is shifted to the left of a number of positions so that it will be in position of bit PORTB5.

For the sake of completeness, we also report here the full schematic diagram of the project, although each individual section also shows its own part in more detail.



## Buttons

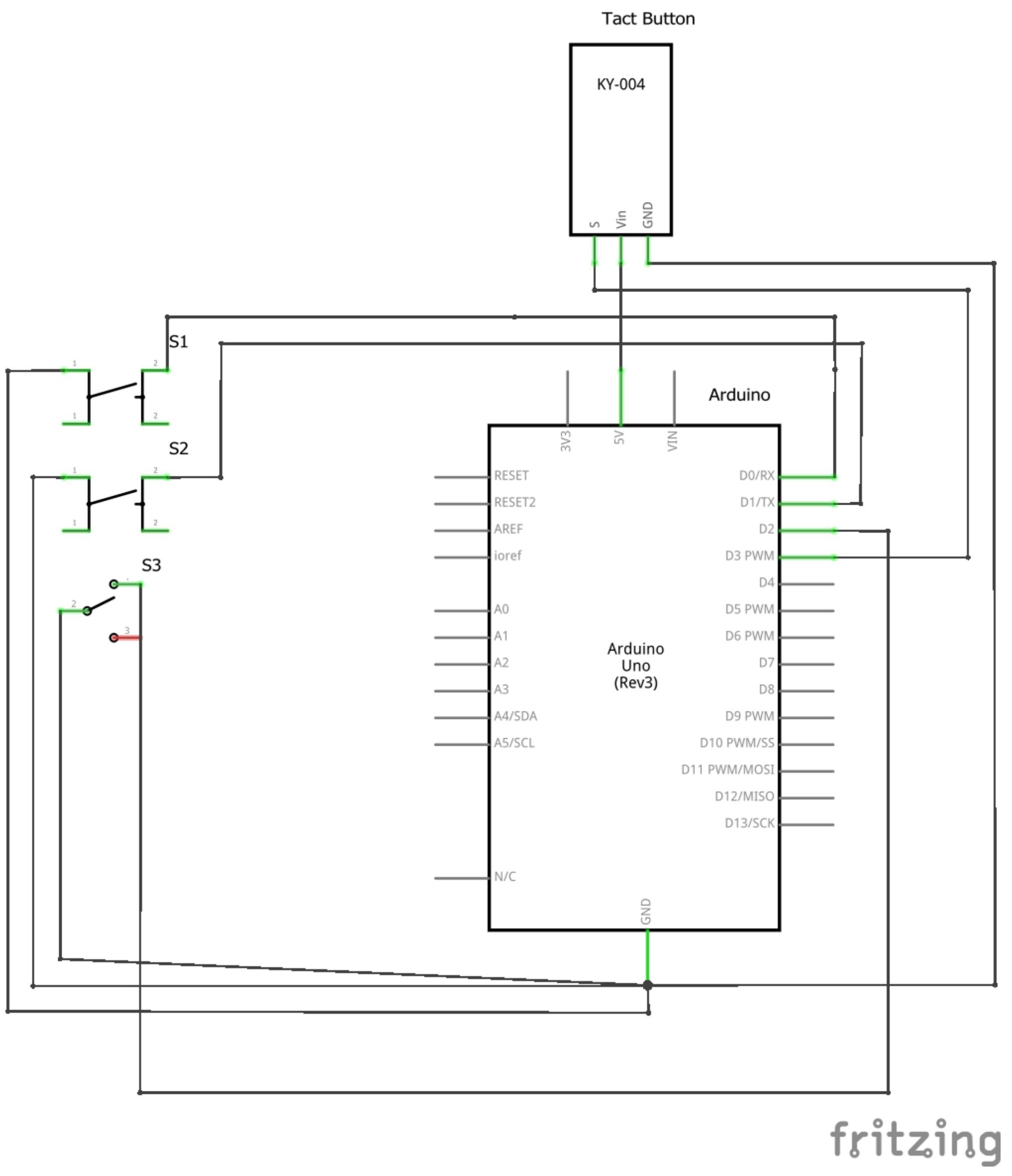
The system involves three different types of buttons.

The right-most black button (BB1) can swap current time with alarm time on the display while the state is IDLS (IDLE State), but it is also used to change the current time or alarm time when we are editing the values. There are two black buttons: in editing mode one (BB1) increases and one(BB2) decreases values by one. BB1 is connected to pin 0 and BB2 to pin 1.

The red button (BR) is used to switch the machine state. It is pressed in order to start editing. This button has two peculiarities: contrary to the others, it is active HIGH, and it execute an action on button release instead of button press. Recording the duration of the pressing, it was possible to define two functionalities: a short press from IDLS state transits the machine to the editing alarm hour, while a long press enables current hour editing. When entered time editing, both time and alarm, another short press on this button switches to minutes editing. Finally, one more press and the system is back to idle state. BR is connected to pin 3.

The DPDT ON-ON switch (BL) is used to enable or disable the alarm, and it’s connected to pin 2.

This is the diagram of how the buttons are wired:



In order to use buttons, pullup resistors are necessary to pull their state to HIGH when inactive, with the exception of BR which features itself a pulldown resistor that keeps its state as LOW when inactive. We are using the internal Arduino pullup resistors.

The black and red buttons are polled. This is the pollButtons() function, called in the loop.

```
void pollButtons() {
  static long debounce[BUTTONS] = {0, 0, 0};
  byte tmpRead;  //temporary value to store button readings
  const byte buttons[] = {BB1, BB2, BR}; // button pins to read
  static byte prevButtonStates[] = {HIGH, HIGH, LOW}; // previous states of
buttons, defaulted to inactive values
  static byte prevProcStates[] = {HIGH, HIGH, LOW}; // previous states of buttons,
defaulted to inactive values
  for (int i = 0; i < BUTTONS; i++) {   //loop each button
    tmpRead = digitalRead(buttons[i]); //read the button pin
    if (tmpRead != prevButtonStates[i]) { //proceed only if the current reading
is different than the previous state of button HIGH->LOW or LOW -> HIGH
      debounce[i] = millis();
      prevButtonStates[i] = tmpRead;    //update the previous button states array
    }
    if (debounce[i] + 25 < millis() && tmpRead != prevProcStates[i]){
      switch (i) {                     //call the appropriate handles passing the
reading
        case 0 : BBHandler(tmpRead, 1); break;
        case 1 : BBHandler(tmpRead, -1); break;
        case 2 : BRHandler(tmpRead); break;
      }
      prevProcStates[i] = tmpRead;
    }
  }
}
```

The prevButtonStates[] array contains the inactive values of the buttons: HIGH for the black ones and LOW for the red one. For each button, the handler is called only if the current reading state is different than the previous processed state and if more than 25 milliseconds passed, in order to debounce the buttons. Buttons oftenly don't change state in a sharp way, but "bounce" between the two states for some time before a more definitive change. Adding a debouncing timer ensures that only one signal is being processed.

Buttons are read using the digitalRead function. The values can also be read using the Arduino register in this way:

```
PIND & (1<<PIND0)
```

```
PIND & (1<<PIND1)
```

```
PIND & (1<<PIND3)
```

Where PIND register contains the readings for PIND0, PIND1 and PIND3 which correspond to pins 0, 1 and 3. To read ony the relevant bit, we apply the bit-wise AND to a 8bit value where we set as '1' only the position where there is the bit we want to read.

### Black button

These buttons are registered on pin 0 and 1:

```
#define BB1 0 //black button 1
#define BB2 1 //black button 2
```

Plus, in the setup():

```
pinMode(BB1, INPUT_PULLUP);
pinMode(BB2, INPUT_PULLUP);
```

This is equivalent to the following register commands:

```
PORTD |= (1<<PORTD0) | (1<<PORTD1)
```

Where a bit set to 1 in the register PORTD positions that correspond to input pins (which is the default) signifies that the pullup resistor is enabled.

During the loop, the poll function calls the BBHandler().

```
void BBHandler(byte reading, char direction){
  if (reading == HIGH)
      return; //ignore HIGH readings
  byte *toEdit;       // byte pointer used for editing
  byte mod;                     //byte variable to do modulo math
  byte fromOne = 0;
  switch (state) {
    case IDLS:
    case RING: showAlarm = (showAlarm + 1) % 2; return; //toggle alarm and current
time, then return
```

```
    case EHRS: toEdit = &timeArray[0]; mod = 24;  break; // the pointer will now
reference the value we want to edit
    case EMIN: toEdit = &timeArray[1]; mod = 60;   break;
    case EAHR: toEdit = &alarmArray[0]; mod = 24;  break;
    case EAMI: toEdit = &alarmArray[1]; mod = 60;  break;
    case EYRS: toEdit = &dateArray[0]; mod = 100; break;
    case EMTH: toEdit = &dateArray[1]; mod = 12; fromOne = 1; break;
    case EDAY: toEdit = &dateArray[2]; mod = 31; fromOne = 1; break;//additional
checks
  }
  if (state == EDAY) {
    switch (dateArray[1]) {
      case 2: !((dateArray[0] + 2000) % 4) && !(!((dateArray[0] + 2000) % 100) &&
((dateArray[0] + 2000) % 400)) ?  mod = 29 : mod = 28; break;
      case 4:
      case 6:
      case 9:
      case 11: mod = 30; break;
      default: break;
    }
  }
  if (direction == -1 && *toEdit == 0){
    *toEdit = mod;
  }else if(fromOne && direction == -1 && *toEdit == 1){
    *toEdit = mod+1;
  }

  *toEdit = (((*toEdit) + direction - fromOne) % mod) + fromOne ; //increase the
value referenced by the pointer and apply the modulo
  noBlink = 1;                    // set the noBlink flag to 1; we don't want to
blink the digits we are editing for a while
  blinkTimer = millis();     // start tracking how much time since the last digit
editing
  displayTime();                    // display the new values immediately, not
waiting till the next interrupt
  displayDate();
}
```

If the system is in the IDLS or RING state, pressing the button BB1 will switch the view from current time to set alarm time and vice versa. On the other hand, if it is in the editing mode, the system must register the press and modify the value of the time/alarm. This is why the byte* toEdit pointer and the variable byte mod are defined. After parsing the current state, toEdit will point the array element needed to be modified (time/alarm, hour/minute, year, month or day) and the value of modulo to be applied. In case of EDAY state, another control is necessary: the maximum number of days depends on the month, so the value of the mod variable is settled according to the value of the month. Notice that also leap years are considered. Prior to editing the value, we also make sure that the value prior to 0 or 1 is the highest possible value, so that by pressing the decrease button the 0th hour is preceded by the 23rd hour and the 12th month is preceded by the 1st month. At this point the value is edited, increased or decreased by one unit according to the pressed button (BB1 or BB2). The new time is immediately displayed. Also, the noblink variable is modified, to avoid continuous blinking during time editing.

## Red Button

The red button is connected to the pin 3.

```
#define BR 3  //red button
```

We don’t need to set a pullup resistor because the button is already active HIGH and features a pulldown resistor by itself.

As defined before, during the loop the pollButton() function calls the BRHandler().

```
void BRHandler(byte reading){
  static long startpressed = 0, endpressed = 0; //timers to count how long it has
been pressed

  if ( reading == HIGH) {
    startpressed = millis();      //button has been pressed; we record the time
with millis()
  }
  else
  {
    if (startpressed + 1000 < millis()) { // if the difference between timings is
over 1000 msec, then it's a long press
      if (state == IDLS) {                  //long presses are valid only if we're
in IDLS state
        timeArray[0] = now.hour();           //we record the current time on the
timeArray; this is the starting value to start editing current time
        timeArray[1] = now.minute();
        dateArray[0] = now.year() - 2000;
        dateArray[1] = now.month();
        dateArray[2] = now.day();
        state = EHRS;
        currentAnimation = ETANIM;
        startAnimation();
      }
    }
    else {
      //state transitions
      switch (state) {
        case IDLS:  state = EAHR; currentAnimation = ETANIM; break;
        case EHRS:  state = EMIN; currentAnimation = ETANIM; break;
        case EMIN:  state = EYRS; currentAnimation = EYRANIM;break; //here we also
write the edited time in timeArray to the RTC clock
        case EYRS:  state = EMTH; currentAnimation = EMTHANIM;break;
        case EMTH:  state = EDAY; currentAnimation = EDAYANIM;break;
        case EDAY:  adjustClock(); state = IDLS; currentAnimation = IDLSANIM;
break;
        case EAHR:  state = EAMI; currentAnimation = ETANIM; break;
        case EAMI:  state = IDLS;  currentAnimation = IDLSANIM; break;
        case RING: return; //RING has no associated transitions
      }
      startAnimation();
    }
  }
}
```

As said before, the red button executes on button release and involves different behaviours according to press duration. For this reason, two variables take track of the length of the duration: startpressed and endpressed record the value of the millis() method, i.e. the milliseconds elapsed since the Arduino was turned on. If the difference between these two variables is bigger than 1000 msec, the user acted a long press and the system switched to the EHRS state. A short press instead involves different behaviours according to the current state. If in IDLS state, the system switches to editing alarm hours (EAHR). From this state, another short press leads switching to editing alarm minutes (EAMI), and from this state another press and the system is back to idle state. If the system is instead in the editing current hour state (EHRS), a short red button press will switch the system to all the editing current datetime: first editing minutes (EMIN), then editing year(EYRS), editing month (EMTH), editing day (EDAY), and another press from this state will call the routine adjustClock()

```
void adjustClock() {
  //DateTime (uint16_t year, uint8_t month, uint8_t day,uint8_t hour =0, uint8_t
min =0, uint8_t sec =0);
  RTC.adjust(DateTime(dateArray[0] + 2000, dateArray[1], dateArray[2],
timeArray[0], timeArray[1], 1));
}
```

This method updates the RTC clock to the new set time and date.

Notice that the RING state has no associated transitions.

Notice that every state involves an animation, saved in the currentAnimation variable. At the end of the controls, the startAnimation() method is called, which affect the animation displayed on the led display.

## Switch (DPDT ON_ON)

The DPDT ON-ON (BL) button is a switch used to enable/disable alarm on the currently set alarm time. In contrast to the others, this button operates with interrupts, as defined in the setup (more information on interrupts are in the relevant section):

```
attachInterrupt(digitalPinToInterrupt(2), BLHandler, CHANGE);
```

We still need to set the pullup resistor in the setup:

```
pinMode(BL, INPUT_PULLUP);
```

or equivalently:

```
PORTD |= (1<<PORTD2)
```

The code of the BLHandler is very simple.

```
void BLHandler() {
  if (digitalRead(BL) == LOW) {
    armed = true;
  }
  else {
    armed = false;
  }
}
```

If the system reads a the switch with LOW position, the global variable armed is set 1, otherwise it is set 0. This variable is used by the chackAlarm() method called in the loop, in order to establish whether the system must be switched to the RING state.

Since it is a Double Throw position, this switch closes the circuit in the UP position as well as the Down position (therefore it is defined ON-ON). Though, in the system it is wired as a normal ON-OFF button.

The value of the button could also be read by direct register access using:

```
PIND & (1<<PIND2)
```

## RTC clock

A real-time clock (RTC) is a computer clock (most often in the form of an integrated circuit) that keeps track of the current time.[3]

Although keeping time can be done without an RTC, using one has benefits:

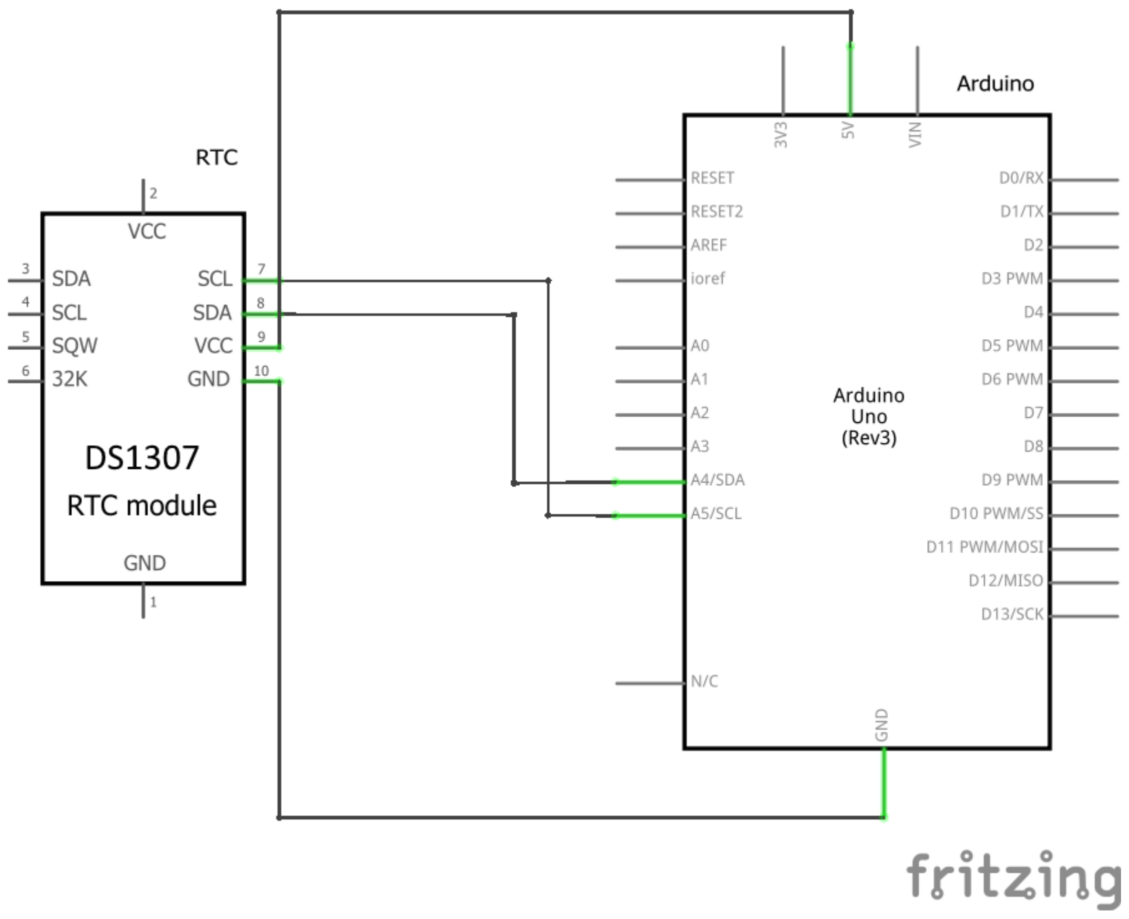
- Low power consumption (important when running from alternate power)
- Frees the main system for time-critical tasks
- Sometimes more accurate than other methods

RTCs often have an alternate source of power, so they can continue to keep time while the primary source of power is off or unavailable. This alternate source of power is normally a lithium battery.

The RTC uses a crystal oscillator with a frequency of 32.768 kHz. This is the same frequency used in quartz clocks and watches, and for the same reasons, namely that the frequency is exactly $2^{15}$ cycles per second, is a convenient rate to use with simple binary counter circuits.

Many commercial RTC ICs are accurate to less than 5 parts per million. In practical terms, this is good enough to perform celestial navigation, the classic task of a chronometer.

The DS1307 RTC device is a real time clock able to count seconds, minutes, hours and date (day, day of the week, month and year). The end of the month date is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The DS1307 has a built-in power sense circuit that detects power failures and automatically switches to the battery supply.



[3] https://en.wikipedia.org/wiki/Real-time_clock

Libraries used to interact with the device are RTClib.h[4] and Wire.h[5] (for the I2C protocol).

On a low level, we setup hardware I2C in the Master receiver mode in this way:

```
TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN)
```

In the register we set bit TWEN to enable the I2C interface, TWSTA to sent the "start" signal, and by writing 1 in TWINT we cleared the interrupt associated with I2C, marking the start of the transmission of the "start" signal.

We then need to wait for the transmission to be complete:

```
while(!(TWCR & (1<<TWINT))
```

When the start signal has been transmitted, the TWINT bit is again set to 1.

We can check for errors by:

```
If((TWSR & 0xF8) != 0xF8)
```

If all went well, we need to set the Master receiver mode by sending the SLA+R signal:

```
TWDR = SLA_R
```

```
TWCR = (1<<TWINT) | (1<<TWEN)
```

We write the SLA_R signal in TWDR, then we enable the I2C interface again and clear the I2C interrupt to start the transmission.

We wait again and then check for errors:

```
while(!(TWCR & (1<<TWINT))
```

```
If((TWSR & 0xF8) != 0x40)
```

We repeat this instruction again:

```
TWCR = (1<<TWINT) | (1<<TWEN)
```

And wait for data from the slave RTC

```
While(!(TWCR & (1<<TWINT)))
```

The data can be read once the bit in TWINT is set from the TWDR register (1 byte):

```
Data = TWDR
```

These last 3 instructions are repeated for each byte. We clear the interrupt, wait for it to be set again and then read the new data. Finally, we stop the transmission by sending the stop signal:

```
TWCR = (1<<TWINT) | (1<<TWEN) | (1<< TWSTO)
```

In the setup the RTC is initialized and set with compilation datetime

```
Wire.begin();                          // initialize the i2c library
RTC.begin();                           // start the clock
```

[4] https://github.com/adafruit/RTClib
[5] https://www.arduino.cc/en/Reference/Wire

```
 RTC.adjust(DateTime(__DATE__,  __TIME__));  //initialize  the  clock  with  the
compilation date and time
```

It is possible to retrieve RTC values by the variable DateTime now. In the loop this variable is continuously updated

```
now   =   RTC.now();   //record   the   current   time   from   the   clock
```

in order to store the current datetime. This variable is used in the diaplayTime() and displayDate() methods in order to show the current values of time and date.

Another crucial function to interact with RTC is the method adjustClock()

```
void adjustClock() {
  //DateTime (uint16_t year, uint8_t month, uint8_t day,uint8_t hour =0, uint8_t
min =0, uint8_t sec =0);
  RTC.adjust(DateTime(dateArray[0]   +   2000,   dateArray[1],   dateArray[2],
timeArray[0], timeArray[1], 1));
}
```

It is used in the red button handler. After setting the new datetimeand saving it in the time_array, the RTC clock is uploaded with these values.

## 7 Segments LED display

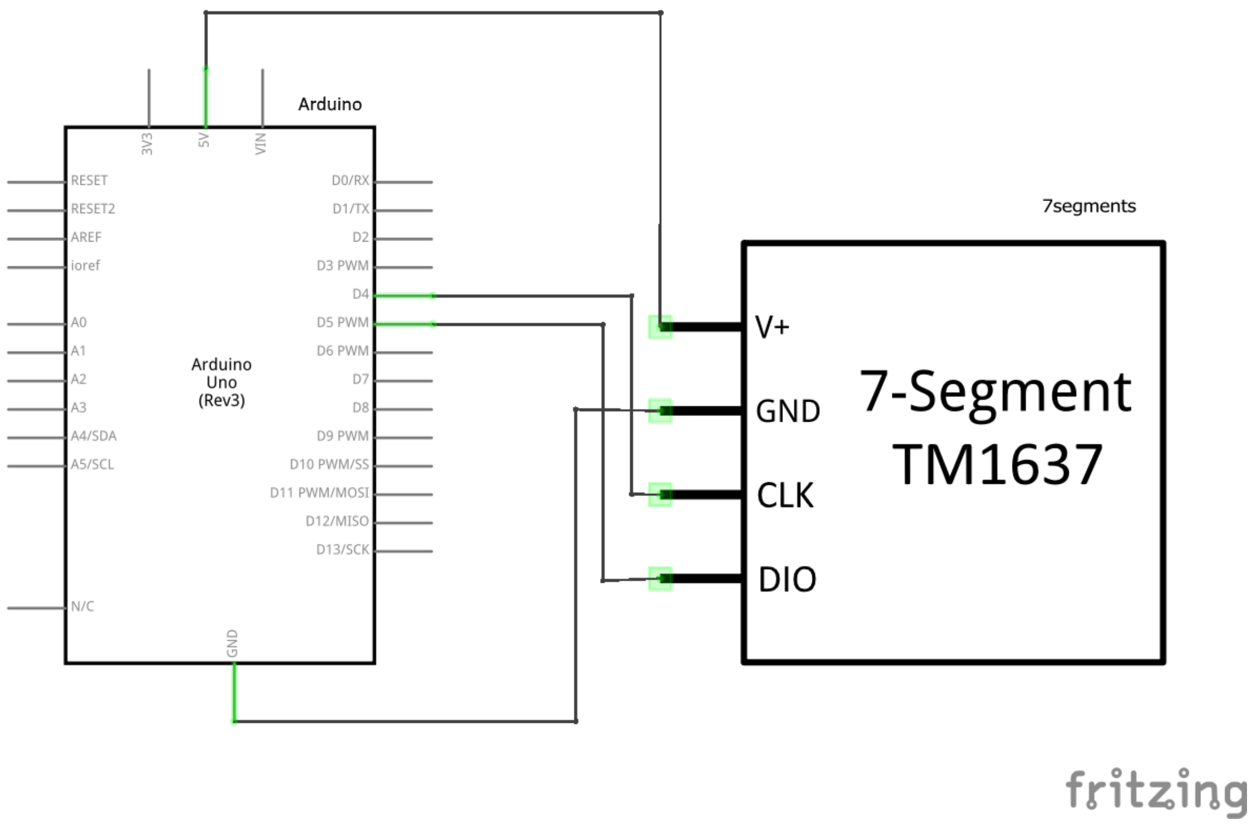
The led display is a 7-segment display with four digits and a colon. It is able to display time and date. It is controlled by the TM1637 chip, which is a LED and key controller.[67] It features:

- controller for six digit of 7-segments display(common anode)
- adjustable brightness
- auto blanking
- scan for 2 x 8 keys
- sychronous interface, 2-wires data + clock

The *7-segment display*, consists of seven LEDs (hence its name) arranged in a rectangular fashion as shown. Each of the seven LEDs is called a segment because when illuminated the segment forms part of a numerical digit (both Decimal and Hex) to be displayed. An additional 8th LED is sometimes used within the same package thus allowing the indication of a decimal point, (DP) when two or more 7-segment displays are connected together to display numbers greater than ten.

This is the wiring diagram:

[6] https://www.electronics-tutorials.ws/blog/7-segment-display-tutorial.html
[7] https://playground.arduino.cc/Main/TM1637



In order to interact with the display, two libraries are included: *SevenSegmentTM1637*.h and *SevenSegmentExtended.h*[8].

The display is connected to pin 4 and 5, respectively CLOCK and DIO and it uses a custom protocol similar to I2C but different, without a slave address. The manifacturer provided a C implementation that lets us show what the libraries are doing under the hood[9]:

The execution flow of the following functions is an initial call to I2CStart(), followed by a series of calls to I2CwrByte() and I2Cask for ACK signals, then concluding with a call to I2CStop(). We edited the code to show it working for the actual pins we are using. We are also assuming that the pins are already set as output.

```
/ / / =====================================
void I2CStart (void) // 1637 start
{
   clk = 1;           // PORTD |= (1<<PORTD4)
   dio = 1;           // PORTD |= (1<<PORTD5)
   Delay_us (2);
   dio = 0;            // PORTD &= ~ (1<<PORTD5)


}
/ / / ============================================
void I2Cask (void) // 1637 Answer
{
   clk = 0;           // PORTD &= ~ (1<<PORTD4)
```

[8] https://github.com/bremme/arduino-tm1637
[9] http://www.microcontroller.it/english/Tutorials/Elettronica/componenti/TM1637.htm

```
  Delay_us (5); // After the falling edge of the eighth clock delay 5us, ACK signals the beginning of judgment
  while (dio);    // while(PIND & (1<<PORTD5))
  clk = 1;        // PORTD |= (1<<PORTD4)
  Delay_us (2);
  clk = 0;        //PORTD &= ~ (1<<PORTD4)
}
/ / / ========================================
void I2CStop (void) // 1637 Stop
{
  clk = 0;        //PORTD &= ~ (1<<PORTD4)
  Delay_us (2);
  dio = 0;        //PORTD &= ~ (1<<PORTD5)
  Delay_us (2);
  clk = 1;        //PORTD |= (1<<PORTD4)
  Delay_us (2);
  dio = 1;        //PORTD |= (1<<PORTD5)
}


/ / / =========================================
void I2CWrByte (unsigned char oneByte) // write a byte
{
  unsigned char i;
  for (i = 0; i <8; i + +)
  {
    Clk = 0;                 //PORTD &= ~ (1<<PORTD4)
    if (oneByte & 0x01) // low front
    {dio = 1;}               //PORTD |= (1<<PORTD5)
    else {dio = 0;}          //PORTD &= ~ (1<<PORTD5)
    Delay_us (3);
    oneByte = oneByte >> 1;
    clk = 1;                 //PORTD |= (1<<PORTD4)
    Delay_us (3);
  }
}
```

The whole behaviour of the display is basically managed by an interrupt. Every 0.5 sec the method displayTime() is executed. This method contains a switch to define the behaviour of the display in every machine state. Thus, the display is used to show hours and minutes, let it be the current time or the alarm time.
In IDLS and RING state the 7-segments shows either the current time or the alarm time, according to the pressure of the black button. Moreover, displaying time the colon will blink thanks to the variable int status.

```
void displayTime() {
  static byte status = 0; //used for blinking, if it's 0 - the selected segments are off, when 1 the segments are on
  switch (state) {
    // in IDLE state, we either show the current time or the alarm time, depending on the showAlarm flag
    // in current time view, the semicolon is blinking using the status variable
    case IDLS:
    case RING:
      if (showAlarm) {
        display.printTime(alarmArray[0], alarmArray[1], true);
      } else {
        display.printTime(now.hour(), now.minute(), status);
      }
      break;
 [...]
}
```

In hours editing states (EAHR, EHRS) the 7-segments shows respectively the time or the alarm time, blinking the first two digits. As example, here is the code of the EAHR state:

```
[...]
   case EAHR:
      display.printTime(alarmArray[0], alarmArray[1], true);
      if (status && !noBlink) {
        display.printRaw((uint8_t)OFF, 0);
        display.printRaw((uint8_t)OFF, 1);
      }
      break;
[...]
```

As the code shows, in this case both status and noblink variable are used to perform digit blinking. The status variable is commuted every time at the end of the function, while the noblink function is modified in the loop. The variable blinkTimer holds the time of the last blinking, while the variable curr represent the current moment: if the difference among these two values is more than 1000, it must blink again.

```
[...]
 if (blinkTimer + 1000 < curr){
    noBlink = 0;
  }
[...]
```

Status EAMI and EMIN for time and alarm minutes editing work like the previous states, but in these cases the third and forth digits are blinking instead.

## Interrupts

An interrupt is a signal that can interrupt the normal execution flow to take care of an immediate event.

In the project we make great use of interrupts. Arduino Uno features many different types of interrupts, but we mostly use only 2 types: external interrupts and timed interrupts.

The external interrupts[10] are associated to a specific pin and there are only 2 of them on the Uno board, associated with pins 2 and 3. The interrupts can be triggered in one of these different cases:

- When the pin is LOW (LOW)
- Whenever the pin changes value (CHANGE)
- When the pin goes from LOW to HIGH (RISING)
- When the pin goes from HIGH to LOW (FALLING)

We use an external interrupt on pin 2 to handle the DPDT switch. It uses a CHANGE interrupt, because it is associated with a flag that will be toggled every time it is switched. It is generally a terrible idea to use interrupts on buttons, because debouncing issues may trigger the interrupt many times and in unwanted ways. But in this case the button is used only to toggle a flag, so even if the button “bounces” its value some times it doesn’t matter; it’s the long term value that will be picked by the code. Other buttons instead could give a series of inputs that, for example, will increase or decrease the value being modified by a lot.

[10] https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

In order to set an external interrupt, we need to perform the following register operations:

```
EICRA |= (1<<ISC00)
```

With this command we set the type of interrupt on the register EICRA, which controls interrupts on the aforementioned pin. Writing “01” on ISC0n we are setting the CHANGE interrupt.

```
EIMSK |= (1<<INT0)
```

This instruction tells the MCU that we want to enable interrupts on pin 2.

We can now write Interrupt Service Routines (ISR) using this syntax:

```
ISR(INT0_vect){...}
```

In fact, the Arduino library has its own functions to deal with interrupts, that summarize all the previous instructions:

```
attachInterrupt(digitalPinToInterrupt(2), BLHandler, CHANGE);
```

This instruction is in the setup() function of the source code.

The other type of interrupts are timed interrupts.[11] These interrupts are triggered when the associated Timer fulfills a certain condition:

- The timer has overflown
- The timer reached a specified value
- An external event happened and the timer returns its value

We are using Timer1, a 16 bit timer, to trigger an interrupt every 500ms. We use this interrupt to refresh the hour and minute value on the 7-segment displays and to eventually blink digits on it. In order to do so we are using a library called TimerOne, but by reading the source we see that the library uses the following register settings[12]:

```
TCCR1B |=  (1<<WGM13) | (1<<CS12)
```

With this instruction we are setting the operating mode of the Timer as “PWM, Phase and Frequency correct” and the relevant thing of this mode is that we can set programmatically the upper limit of the timer, instead of the default 65535 allowed by the 16bits. The second part of the instruction sets the prescaler to 256, which will slow down the timer ticks.

The register that holds the upper limit is ICR1, which will be set at 31250 (it is a 16bit register), according to the following formula:

$$\frac{1}{ClockFrequency} \cdot prescaler \cdot ICR1 = \frac{1}{16000000} \cdot 256 \cdot 31250 = 0.5s$$

Lastly, we set Timer1 to trigger an interrupt on overflow with:

```
TIMSK1 |= (1<<TOIE1)
```

We can now write the ISR using this syntax:

```
ISR(TIMER1_OVF_vect){ . . . }
```

[11] https://www.teachmemicro.com/arduino-timer-interrupt-tutorial/
[12] http://www.gammon.com.au/interrupts

Using the library we only need to call these 2 functions:

```
Timer1.initialize(500000);
Timer1.attachInterrupt(displayTime);
```

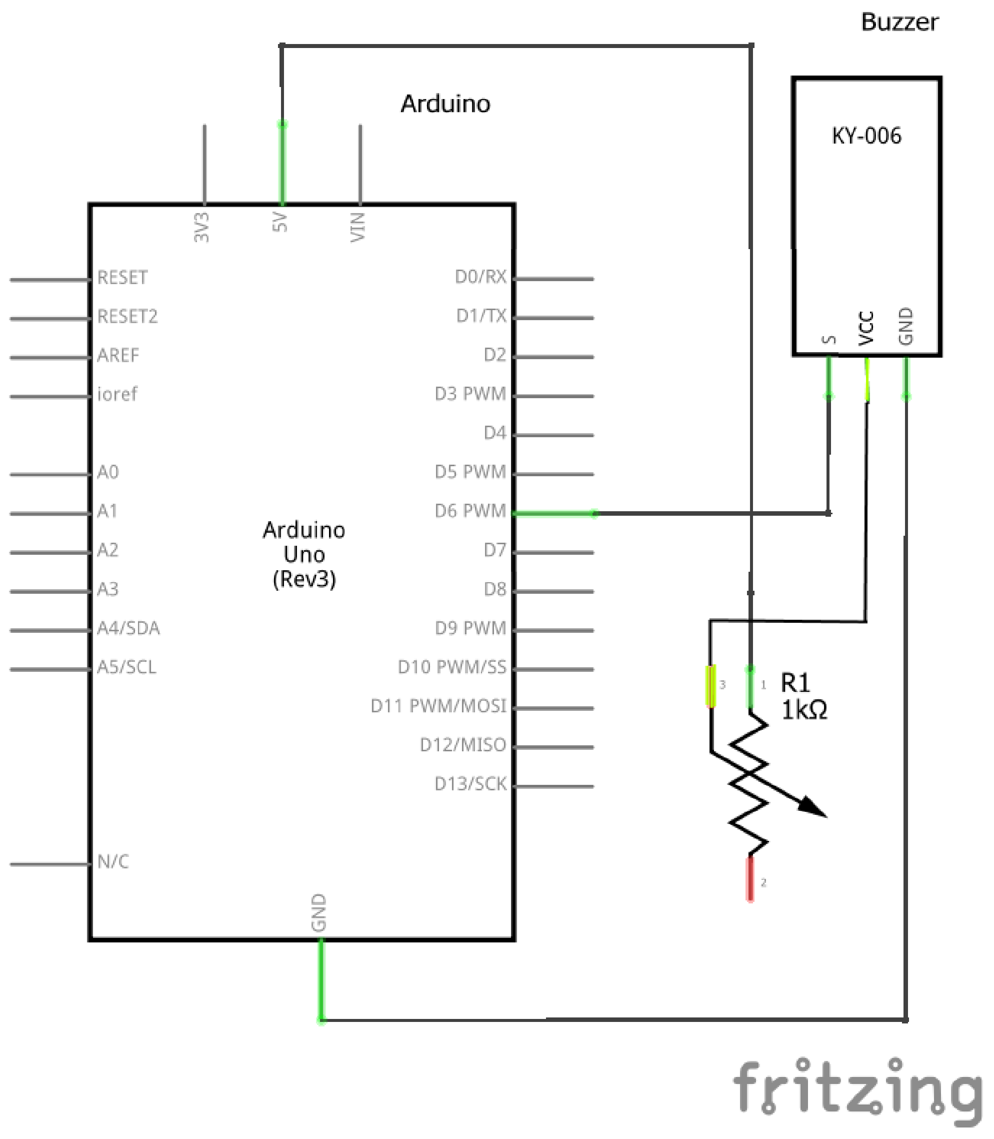
The first one sets the period of the interrupt at 500.000 microseconds, or 0.5 seconds. The second function sets the ISR associated with the interrupt.

## Buzzer

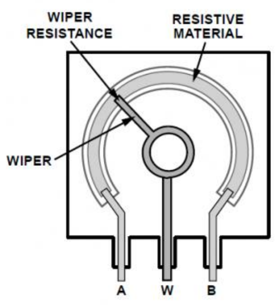
The buzzer is a piezoelectric device used to play the alarm. We use a passive buzzer because it is able to make different tones, but the devices that controls the buzzer has to provide it with an oscillating electronic signal at a desired frequency. The supplied frequency will determine the tone. Supplying just a fixed voltage will generate no sound, except perhaps a slight "tick" at the point when the power source is connected or disconnected from the buzzer.[13][14]

In the system the buzzer is connected to the pin 6 of the Arduino, and gets its voltage through a potentiometer.

[13]https://github.com/fochica/fochica-wiki/wiki/Passive-buzzer-guide
[14] https://electronics.stackexchange.com/questions/224374/active-vs-passive-buzzer



In order to regulate the sound intensity, the system is equipped with a potentiometer.



It is a variable resistor. By rotating the knob, it is possible to increase or decrease electric voltage. In this way the sound played by the buzzer will be louder or lower. The maximum resistance is $1k\Omega$.

The buzzer is used to play a melody when the alarm goes off.

The melodies are stored in these 2 arrays, the first of which contains the note frequencies and the second contains the duration. The definition of these macros is in the included file pitches.h.

```
const int melody[] = {
    NOTE_C4, NOTE_D4, NOTE_F4, NOTE_D4, NOTE_A4, PAUSE, NOTE_A4, NOTE_G4, PAUSE,
NOTE_C4,  NOTE_D4,  NOTE_F4,  NOTE_D4,  NOTE_G4,  PAUSE,  NOTE_G4,  NOTE_F4,  PAUSE,
PAUSE, PAUSE
};

//the length of each note
const  byte  noteDurations[]  =  {EIGHT_NOTE,  EIGHT_NOTE,  EIGHT_NOTE,  EIGHT_NOTE,
EIGHT_NOTE, EIGHT_NOTE, HALF_NOTE, HALF_NOTE, QUARTER_NOTE, EIGHT_NOTE,
                                EIGHT_NOTE,  EIGHT_NOTE,  EIGHT_NOTE,  EIGHT_NOTE,
EIGHT_NOTE, HALF_NOTE, HALF_NOTE, QUARTER_NOTE, QUARTER_NOTE, QUARTER_NOTE
};
```

Playing is performed by the playMelody() method. It is called by the checkAlarm() method executed in the loop, only if the current state is RING.

```
void playMelody() {
  static long notePause = 0, betweenPause = 0;
  static bool isPlaying = false, isWaiting = false;
  int noteDuration,pauseBetweenNotes,actualPause,curr;

   if (!isPlaying){
        noteDuration = 1000 / noteDurations[currentNote];  //length of the note
        pauseBetweenNotes = noteDuration * 1.30;         //calculating an adequate
pause prior the next note
        actualPause = pauseBetweenNotes - noteDuration;
        curr = melody[currentNote];
    if (curr != PAUSE)
      tone(BUZZER, melody[currentNote], noteDuration);      // playing the note
on the buzzer pin
    notePause = millis();
    isPlaying = true;
  }

  if (notePause + noteDuration > millis())
      return;

  noTone(BUZZER);
  digitalWrite(BUZZER, HIGH);

  if (!isWaiting) {
    betweenPause = millis();
    isWaiting = true;
  }

  if (betweenPause + actualPause > millis())
      return;


  currentNote = (++currentNote) % MELODY_LENGTH;    //change index to the next
note (the melody will replay using modulo math)
  isPlaying = false;
  isWaiting = false;

}
```

This method uses two global arrays: melody[] contains the notes to be played, while noteDurations[] contains the duration of each note. These arrays must me of the same size, since each note must have its duration. This method executes one note per time, tracked by the currentNote variable. Therefore, it must be called several times in order to play a complete melody. After retrieving the current note to execute and its duration, if this note is not a PAUSE, it is played. Following every note there is a short pause, realized by the variables notePause and betweenPause with the help of isPlaying and isWaiting flags; the first pause makes sure we wait for the note to finish playing, while the second makes sure we observe a short pause between each note.  Next the currentNote value is increased by one. Thanks to the modulo function, after the last note the first one will start again.

On a low level, tunes are played this way[15][16]:

First we have to initialize the 8bit timer known as Timer2.

```
TCCR2A = 0
```

```
TCCR2B = 0
```

```
TCCR2A |= (1<<WGM21)
```

We reset the TCCR2A and TCCR2B registers, then we set the bit WGM21 to put the timer in CTC mode:

> *"In Clear Timer on Compare or CTC mode (WGM2[2:0] = 2), the OCR2A Register is used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT2) matches the OCR2A. The OCR2A defines the top value for the counter, hence also its resolution. This mode allows greater control of the compare match output frequency. It also simplifies the operation of counting external events."*

Then, we set pin 6 as output and we set the appropriate clock prescaler, which in our case is, for all the frequencies that we are currently using, 128.

```
DDRD |= (1<<DDRD6) // pin 6 as output
```

```
TCCR2B |= (1<<CS22) | (1<<CS20) // prescaler
```

We need to calculate the value to write in OCR2A, this is done by this formula (example for a tune of frequency 294hz:

$$ocr = F_{CPU}/\ frequency\ /2/128\ \ -1 = 16000000/\ 294\ /2/128\ -1 = 211$$

So we write it into OCR2A 8bit register:

```
OCR2A = ocr
```

Then, we calculate a toggle_count value, used to to turn off the tune when the elapsed time for its duration has passed. This is calculated as (example for frequency 294hz and duration 300ms):

$$toggle\_count = 2 \cdot frequency \cdot duration/1000 = \ 2 \cdot 294 \cdot 300/1000\ \ = 176$$

Lastly, we need to set the interrupt mode:

```
TIMSK2 |= (1<<OCIEA)
```

The mode is Output Compare A Match. An interrupt is generated whenever the timer value is equal to the value in OCR2A register, and since we are in CTC mode the timer is cleared at that point too.

The ISR is written in the following way:

```
  ISR(TIMER2_COMPA_vect)
```

[15] https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/Tone.cpp
[16] https://www.avrfreaks.net/forum/avr-digitalpintobitmask-arduino-macro

```
{

  if (timer2_toggle_count != 0)
  {
    // toggle the pin
    *timer2_pin_port  ^=  timer2_pin_mask;  //timer2_pin_port  is  a
pointer to PORTD, timer2_pin_mask is the value of the bit we want to
change
    // also equivalent to alternatively call:
     PORTD |= (1<<PORTD6)
    //and
    PORTD &= ~(1<<PORTD6)

    if (timer2_toggle_count > 0)
      timer2_toggle_count--;
  }
  else
  {
    // need to call noTone() so that the tone_pins[] entry is reset,
so the
    // timer gets initialized next time we call tone().
    // XXX: this assumes timer 2 is always the first one used.
    noTone(tone_pins[0]);
  }
}
```

So if the toggle_count variable is not 0, the value on pin 6 is toggled and toggle_count is decreased. When it is 0, the call to noTone turns the buzzer off.

The call to noTone does the following operations:

`TIMSK2 &= ~(1<<OCIEA)`

This instruction clears the OCIEA bit, thus disabling the interrupt.

`TCCR2A = (1<<WGM20)`

With this we reset the register to the operating mode PWM and Phase correct.

`TCCR2B = (TCCR2B & 0b11111000)`

And with this we reset the prescaler to the default 0.

Finally, we reset the OCR2A register:

`OCR2A = 0`

And write LOW on pin 6:

`PORTD &= ~ (1<<PORTD6)`

Then in our code we need to put that pin HIGH again because our buzzer is active HIGH:

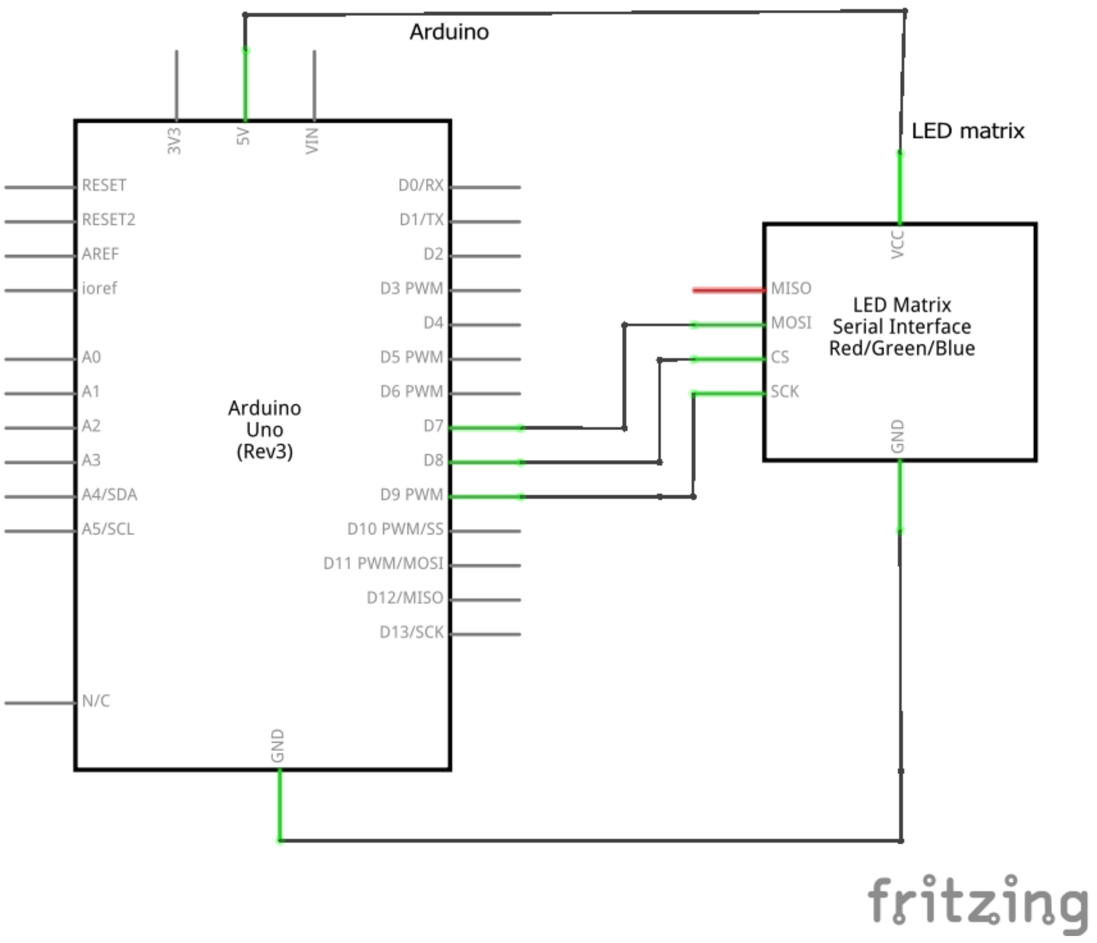
```
PORTD |= (1<<PORTD6)
```

## LED matrix

The LED matrix component features 64 red LEDS controlled by the MAX7219 chip, which uses SPI communication protocol.[17]

The MAX7219 is a compact, serial input/output common-cathode display driver that interfaces microprocessors (µPs) to 7-segment numeric LED displays of up to 8 digits, bar-graph displays, or 64 individual LEDs. Included on-chip are a BCD code-B decoder, multiplex scan circuitry, segment and digit drivers, and an 8x8 static RAM that stores each digit. Only one external resistor is required to set the segment current for all LEDs. The MAX7221 is compatible with SPI™, QSPI™, and MICROWIRE™, and has slew-rate-limited segment drivers to reduce EMI.

The devices include a 150µA low-power shutdown mode, analog and digital brightness control, a scan-limit register that allows the user to display from 1 to 8 digits, and a test mode that forces all LEDs on.



The LedControl library[18] takes care of the SPI communication, but according to the data sheet these are the registers to be set to send data using hardware SPI:

[17] https://www.maximintegrated.com/en/products/power/display-power-control/MAX7219.html
[18] http://wayoda.github.io/LedControl/pages/software

With this line we are setting MOSI (pin 11), CS (pin 8) and CLK (pin 13) as output pins

```
DDRB |= (1<<PB3) | (1<<PB5) | (1<<PB0);
```

With this we are enabling SPI, setting the Master and setting the clockrate.

```
SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR0)
```

This is the procedure to send a byte, named "data".

We set the CS pin as LOW

```
PORTB &= ~(1<<PB0)
```

We write data in the appropriate register

```
SPDR = data
```

Then we are waiting for the transmission to end

```
while(!(SPSR & (1<<SPIF));
```

Finally, we write HIGH back on the CS pin

```
PORTB |= (1<<PB0)
```

The library uses software SPI though, and the device isn't even connect to the hardware SPI pins, so the procedure is actually different:

```
  Byte spidata[2];
  spidata[1]=opcode;
  spidata[0]=data;
```

We setup a byte with the operation code and a byte with the actual data, i.e. which LEDS are to be on and which LEDS are to be off.

```
  PORTB &= ~(1<<PB0)
```

We turn the CS pin low

```
  //Now shift out the data
  for(int i=2;i>0;i--)
    shiftOut(SPI_MOSI,SPI_CLK,MSBFIRST,spidata[i-1]);
```

With this loop we are now sending the data, then finally turn the CS pin high again

```
  //latch the data onto the display
  PORTB |= (1<<PB0)
```

The shiftOut function is actually carrying out the transfer:

```
void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, uint8_t val)
{
        uint8_t i;

        for (i = 0; i < 8; i++)  {
                if (bitOrder == LSBFIRST)
                        digitalWrite(dataPin, !!(val & (1 << i)));
                          //equivalent to write either
                          PORTD |= (1<<PD7) for bit 1
                          or
```

```
                           PORTD &= ~(1<<PD7) for bit 0


                  else
                         digitalWrite(dataPin, !!(val & (1 << (7 - i))));
                           //equivalent to write either
                           PORTD |= (1<<PD7) for bit 1
                           or
                           PORTD &= ~(1<<PD7) for bit 0


                 digitalWrite(clockPin, HIGH); //equivalent to PORTB |= (1<<PB1)
                 digitalWrite(clockPin, LOW);  //equivalent to PORTB &= ~(1<<PB1)
         }
}
```

The LED Matrix is used to show the new state after a state transition. The state is shown as a sequence of 4 8bit images The sequence is repeated 3 times, then the LED matrix is turned off again to limit power consumption.

The sequences are defined in this array:

```
const byte animations[][4][8] = {
  {{0x0, 0xe4, 0xa4, 0xee, 0x84, 0x87, 0x0, 0x0}, {0x0, 0xe4, 0xa4, 0xee, 0x84,
0x87, 0x0, 0xc0}, {0x0, 0xe4, 0xa4, 0xee, 0x84, 0x87, 0x0, 0xd8}, {0x0, 0xe4,
0xa4, 0xee, 0x84, 0x87, 0x0, 0xdb}},
  {{0x3c, 0x42, 0x91, 0x91, 0x9f, 0x81, 0x42, 0x3c}, {0x3c, 0x42, 0x91, 0x91,
0x91, 0x91, 0x52, 0x3c}, {0x3c, 0x42, 0x91, 0x91, 0xf1, 0x81, 0x42, 0x3c}, {0x3c,
0x52, 0x91, 0x91, 0x91, 0x81, 0x42, 0x3c}},
  {{0xae, 0xa8, 0xa8, 0xee, 0x48, 0x48, 0x48, 0x4e}, {0x4e, 0xaa, 0xaa, 0xee,
0xac, 0xaa, 0xaa, 0xaa}, {0x64, 0x24, 0xa5, 0x0, 0xff, 0xff, 0x7e, 0x3c}, {0x88,
0xeb, 0x2a, 0x0, 0xff, 0xff, 0x7e, 0x3c}},
  {{0x0, 0x0, 0x6c, 0x54, 0x44, 0x44, 0x44, 0x0}, {0x0, 0x0, 0x7c, 0x44, 0x44,
0x44, 0x7c, 0x0}, {0x0, 0x0, 0x44, 0x64, 0x54, 0x4c, 0x44, 0x0}, {0x0, 0x0, 0x0,
0xea, 0x4e, 0x4a, 0x4a, 0x0}},
  {{0x0, 0x38, 0x24, 0x24, 0x24, 0x38, 0x0, 0x0}, {0x0, 0x3c, 0x24, 0x3c, 0x24,
0x24, 0x0, 0x0}, {0x0, 0x24, 0x24, 0x3c, 0x4, 0x3c, 0x0, 0x0}, {0x89, 0x5a, 0x24,
0x42, 0x42, 0x24, 0x5a, 0x81}},
  {{0x0, 0x0, 0x24, 0x0, 0x18, 0x0, 0x0, 0x0}, {0x0, 0x0, 0x24, 0x0, 0x3c, 0x0,
0x0, 0x0}, {0x0, 0x0, 0x24, 0x0, 0x3c, 0x24, 0x3c, 0x0}, {0x0, 0x0, 0x24, 0x0,
0x7e, 0x42, 0x7e, 0x0}}
};
```

Each element of the array is an array of 4 arrays of 8 byte values. Each byte value specifies in bit notation which LEDs on that row are to be turned on or off.

```
byte rep = REPS;
byte currentAnimation = IDLSANIM;
void writeByteArray(byte *src){
  for (int i = 0; i < MATRIXLEN; i++) {
    lc.setRow(0, i, src[i]);
  }
}
void startAnimation(){
  rep = 0;
  lc.shutdown(0,false);
}
void animate(byte (*src)[8]){
  static byte curr = 0;
  writeByteArray(src[curr]);
  curr = (++curr) % ANIMLEN;
  if (curr == 0){
    rep++;
  }
}
void loop() {
...
static long sequenceDelay = 0;
if (sequenceDelay + 500 < curr){
    if( rep < REPS){
      animate(animations[currentAnimation]);
      sequenceDelay = curr;
    }
    else{
      lc.shutdown(0,true);
    }
  }
...
}
```

These functions define how an animation is played. The global variable rep is initialized with the macro REPS, which equals to 3; while at this state, no animation will be played.  The loop calls the function animate() with the appropriate sequence to play only if more than half a second has passed and the rep variable is less than 3 (so it means that the sequence has to replayed again). If rep is 3 or more, the LED matrix is set to power saving mode.

The function animate() takes in input a sequence (i.e. an element of the global array animations) and uses a static variable to keep track of the current image in a sequence. When called it will call the writeByteArray() function passing the current image to print the image to the LED matrix, increase its curr variable using modulo arithmetic (so the variable will go back to 0 after printing image 3) and if curr is equal to 0 it means one full sequence has been played and rep variable is increased by one. When rep is again equal to 3, the function won’t be called anymore, until a state transition will call startAnimation() which resets the rep variable to 0 and turns on the LED matrix from power saving mode again.

The writeByteArray uses a library function to write each byte of the array in input in the corrispective row of the matrix.

```
void BRHandler(byte reading){
  ...
      //state transitions
      switch (state) {
        case IDLS:  state = EAHR; currentAnimation = ETANIM; break;
        case EHRS:  state = EMIN; currentAnimation = ETANIM; break;
        case EMIN:  state = EYRS; currentAnimation = EYRANIM;break;
        case EYRS:  state = EMTH; currentAnimation = EMTHANIM;break;
        case EMTH:  state = EDAY; currentAnimation = EDAYANIM;break;
        case  EDAY:  adjustClock();  state  =  IDLS;  currentAnimation  =  IDLSANIM;
break;
        case EAHR:  state = EAMI; currentAnimation = ETANIM; break;
        case EAMI:  state = IDLS;  currentAnimation = IDLSANIM; break;
        case RING: return; //RING has no associated transitions
      }
      startAnimation();
    }
  }
}
```

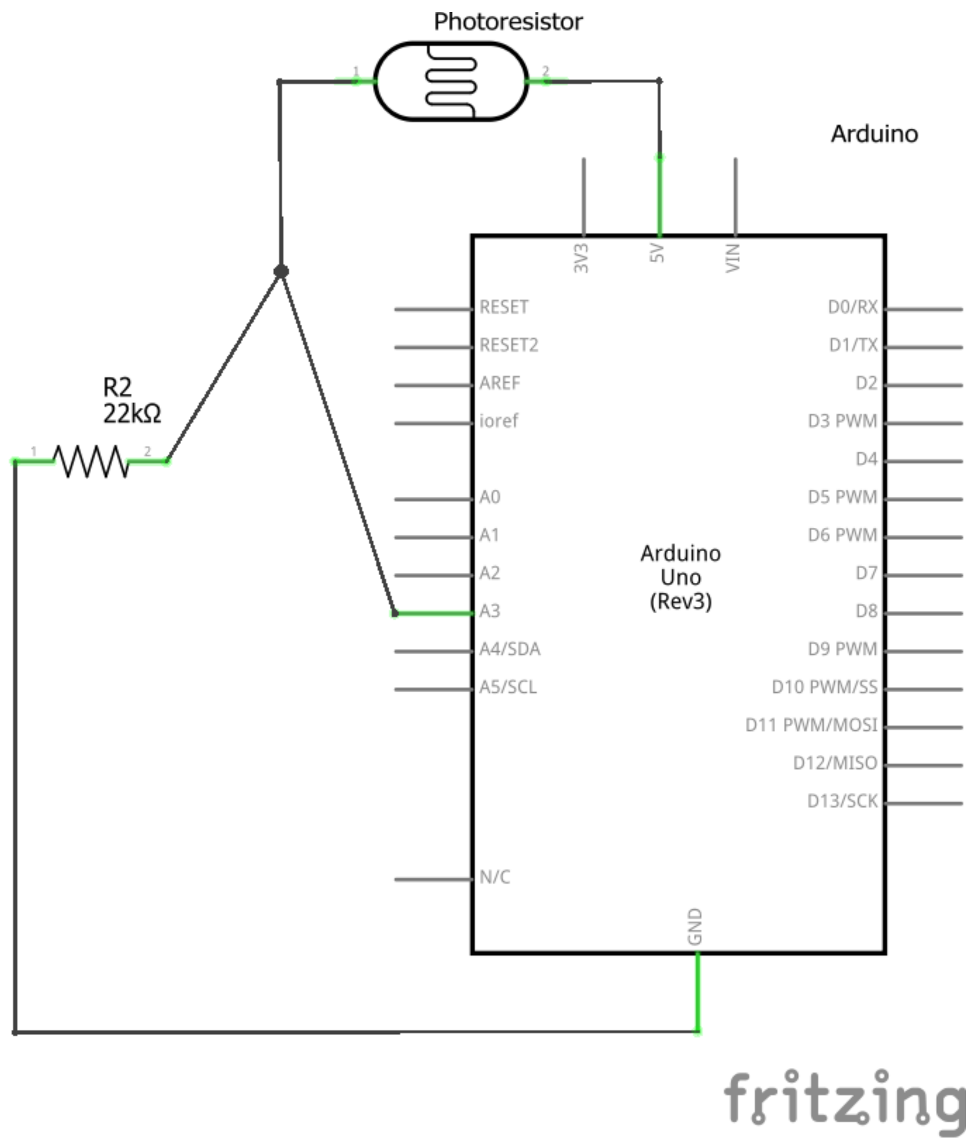
In this piece of code we can see an example of how an animation is triggered ( 2 cases are omitted here: when we are editing the the current hour, from this same function, and when the alarm is started, from the checkAlarm() function): after a state transition in each switch case, we set currentAnimation to the desired animation, then at the end we always call startAnimation().

## Light sensor

A photoresistor, or light dependent resistor, is a type of light sensitive resistor which can measure the intensity of light received. With low light their resistance is high, as high as 600kΩ at 0.1 lux, and with light exposure their resistance drops down, even as low as 100 Ω at 10000 lux. Photoresistors aren't particularly accurate, in fact they are sensitive to temperature changes as well, and the relation between light intensity and resistance is nonlinear, meaning that a slight increase in light would result in a much greater decrease in resistance. Photoresistors also experience latency, with as much as a 1s latency to go from a very strong light to total darkness resistance. Nevertheless, these issues are not a concern for the purpose of our project, in which this component is used to adapt the screens brightness and not to make accurate real time measurements.[19]

The sensor is wired up with an additional 22kΩ resistor so that we can measure the difference in voltage for different resistance values in the photoresistor. The Arduino board in fact reads analog input by mapping the range 0 – 5V in input to 10 bit values, and does not actually read resistance differences. The value of the resistor was determined heuristically by comparing the voltage reading at different exposures of light, in order to have the widest range of values for the light settings that could be produced in the laboratory.

[19] http://www.resistorguide.com/photoresistor/



The code for the photoresistor functionality is contained in the loop() function:

```
void loop() {
  ...
  static long secondPeriod = 0;

 long curr = millis();

  if (secondPeriod + 1000 < curr){
    short phread = analogRead(PHOTORES);
    lc.setIntensity(0, map(phread, 0, 1023, 0, 15)); //0 - 15
    display.setBacklight(map(phread, 0, 1023, 10, 100)); // 0 -100
    secondPeriod = curr;
    ...
 }
 ...
}
```

We can see that by using the millis() function to return the current time in milliseconds and a long variable secondPeriod, we read the resistor value about every 1s. This value is read by performing an analog read on the pin, which can return a value from 0 to 1024. At a low level, this reading can be achieved this way by using the Arduino ADC[20]:

```
ADMUX = (1<<MUX1) | (1<<MUX0);      // ADC3
```

```
ADMUX |= (1 << REFS0);   // use AVcc as the reference
```

The ADMUX register is used to select the input pin by writing the appropriate value on the 5 least significant bits, to select analog pin 3 (ADC3) we just need to write 1 on the MUX0 and MUX1 position (00011 corresponds to ADC3). Then by writing 1 in position REFS0 we are using the reference voltage of 5V.

```
// 128 prescale for 8Mhz
   ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
   ADCSRA |= (1 << ADEN);    // Enable the ADC
   ADCSRA |= (1 << ADSC);    // Start the ADC conversion
```

The ADCSRA is used firstly to set the proper prescaler, 128, by writing 1 in the 3 least significant positions. This prescaler gives the most accurate results, but each reading will take 104$\mu$s.
Then we set 1 in ADEN to enable the ADC circuit and we set 1 in ADSC to start the conversion.

```
while(ADCSRA & (1 << ADSC)); // waits for the ADC to finish
```

This loop is used to wait for the end of the conversion, by checking when the bit ADSC in ADCSRA is set to 1 by the MCU.

```
int ADCval;
ADCval = ADCL;
ADCval = (ADCH << 8) + ADCval;
```

The converted value is read from ADCL and ADCH register, which contain the least significant and most significant bits respectively. ADCH only contains 2 useful bits while the rest are reserved, so we are getting a 10bit return value.

Then using the map function provided by the Arduino library, we are able to map each value in the range 0 to 1023 to the appropriate range used by the display we are regulating the brightness of. The LED matrix has 0 to 15 levels of brightness (with the lowest level being still visible), while the 7-segment displays have a range of 0 to 100, but according to the display library the lowest 10 values mean that the display will be completely off, so we are mapping only from 10 to 100.

## Temperature & Humidity sensor

The DHT11 component is a temperature and humidity sensor.

The temperature is sensed using a thermistor component whose resistance value changes according to the temperature it is exposed to. The resistance is then measured by passing direct current through it and recording the voltage drop produced. The component is in fact an electrical resistor that is additionally sensitive to temperature.[21]

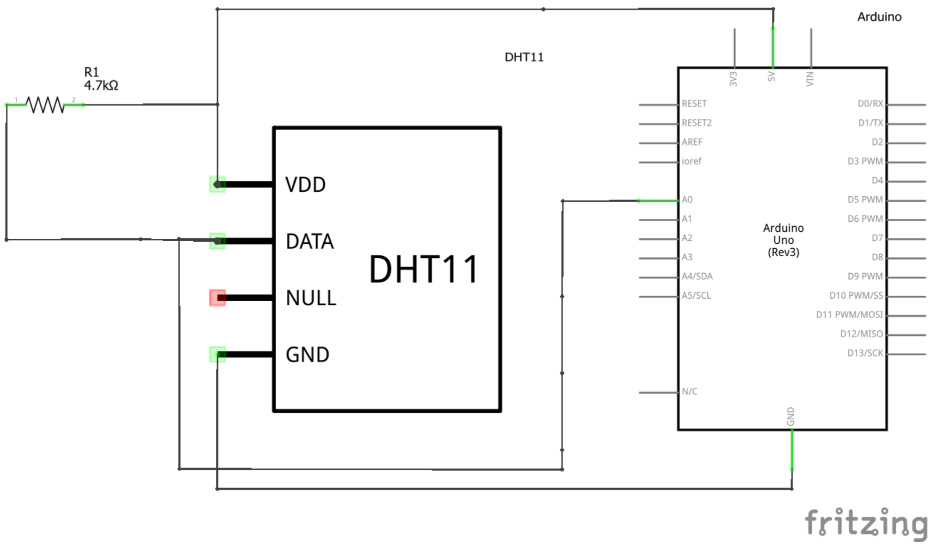
[20] https://www.gammon.com.au/adc

[21] https://www.ametherm.com/thermistor/what-is-an-ntc-thermistor

The relative humidity is sensed using a capacitive humidity sensor. A small capacitor composed of hygroscopic dielectric material is placed between a pair of electrodes. At normal room temperature the dielectric constant of water vapor is about 80, much larger than the constant of the dielectric material (ranging from 2 to 15). Therefore, the absorption of moisture by the sensor results in increased sensor capacitance and relative humidity, which depends on ambient temperature and water vapor pressure, can be calculated using the relationship there exists with the amount of moisture in the sensor and sensor capacitance.[22]

This sensor has a quite limited accuracy. The humidity can be measured only in the 20% to 90% range with an accuracy of $\pm 5\%$ while the temperature has a range of 0°C to 50°C and an accuracy of $\pm$2°C. These values are nonetheless acceptable for a consumer-grade alarm clock application. The sensor also has a sampling rate of 1Hz, so it cannot be polled more than once per second.

The following schematic shows how the component is wired up in the project. A pullup resistor of $4.7k\Omega$ is applied to the DATA pin as the datasheet requires.



The sensor uses a serial interface called Single-Wire Two-Way. A single data line is used to send commands and to read responses. The whole communication process takes about 4ms.

The sensor sends the data in the following format:

| | |
|---|---|
| 8 bit | Integral part of humidity |
| 8 bit | Decimal part of humidity |
| 8 bit | Integral part of temperature |
| 8 bit | Decimal part of temperature |
| 8 bit | Checksum |

The DHT11 sensor is in low power mode unless it is transmitting a message. The MCU signals the sensor that he wants to read the data by sending a LOW voltage level on the DATA line, which when inactive is HIGH thanks to the pullup resistor, that must last at least 18ms. Then, the MCU must wait 20-40 $\mu$s for the sensor's response. The DHT response is an 80$\mu$s LOW signal followed by an 80$\mu$s HIGH signal. The data transmission

[22] https://www.rotronic.com/en/humidity_measurement-feuchtemessung-mesure_de_l_humidite/capacitive-sensors-technical-notes-mr

then begins. Every bit is preceded by a 50$\mu$s LOW signal, then the length of the following HIGH signals determines whether the bit is “0” or “1”. A 26-28 $\mu$s signal is a “0” while a 70$\mu$s signal is a “1”. At the end of the transmission the sensor sets the voltage LOW for 50$\mu$s, then the pullup resistor will reset it to HIGH and the channel is free again.

The author of the SimpleDHT library[23] we use in the project found out that in practice the aforementioned timings aren't accurate, so in his code they are different. The original timings are still specified as comments.

In order to read from pin A0(PC0), the Arduino library performs the following operation:

```
Value = PINC & (1<<PINC0)
```

All pins are input by default, so we only need to read from the register PINC the value in position PINC0. In this case the value variable will have 0 in case of a LOW signal and a non-zero value (128) in case of a HIGH signal.

In our project, we read from the sensor relatively seldom, a rate of just one time per minute. The reason is twofold: one reason is that reading from the sensor implies using short delays (unless we wanted to write our own library that is) and these are to be avoided as much as possible, because the MCU won't be responsive during the delay; the other reason is that there is no point in reading the values so often on a consumer-grade product when, unless solicited on purpose, changes in temperature and humidity happen so slowly.

```
void loop() {
  static long secondPeriod = 0;
  static bool screenUpdated = false;
  now = RTC.now();
  byte second = now.second();
  long curr = millis();
  ...

 if (secondPeriod + 1000 < curr){
    ...
   if (second > 0) {
      screenUpdated = false;
    }
  }
  if (screenUpdated == false && second == 0){
    ...
    readDHT();
    screenUpdated = true;
  }
}
```

In the loop() function, we read the data from the sensor with the call to readDHT(). This call is executed when the second of the current time is 0, i.e. at 16:24:00, then 16:25:00 and so on. Of course, the function could be executed more than once in one second, so we set a flag screenUpdated to true right after we poll the sensor, and reset the flag to false once the second is no more 0, thus ensuring that the sensor won't be polled more than once per minute. There isn't much use in reading temperature and humidity very often, and the sensor is relatively slow. The flag is reset inside a piece of code that is executed at most once per second. It could have been reset everywhere in the loop() code but this way the if clause is executed at most 59 times per minute, one of which will evaluate as true and reset the variable (at second = 1), instead of thousands of time, saving a little bit of processing time.

[23] https://github.com/winlinvip/SimpleDHT/blob/master/SimpleDHT.h

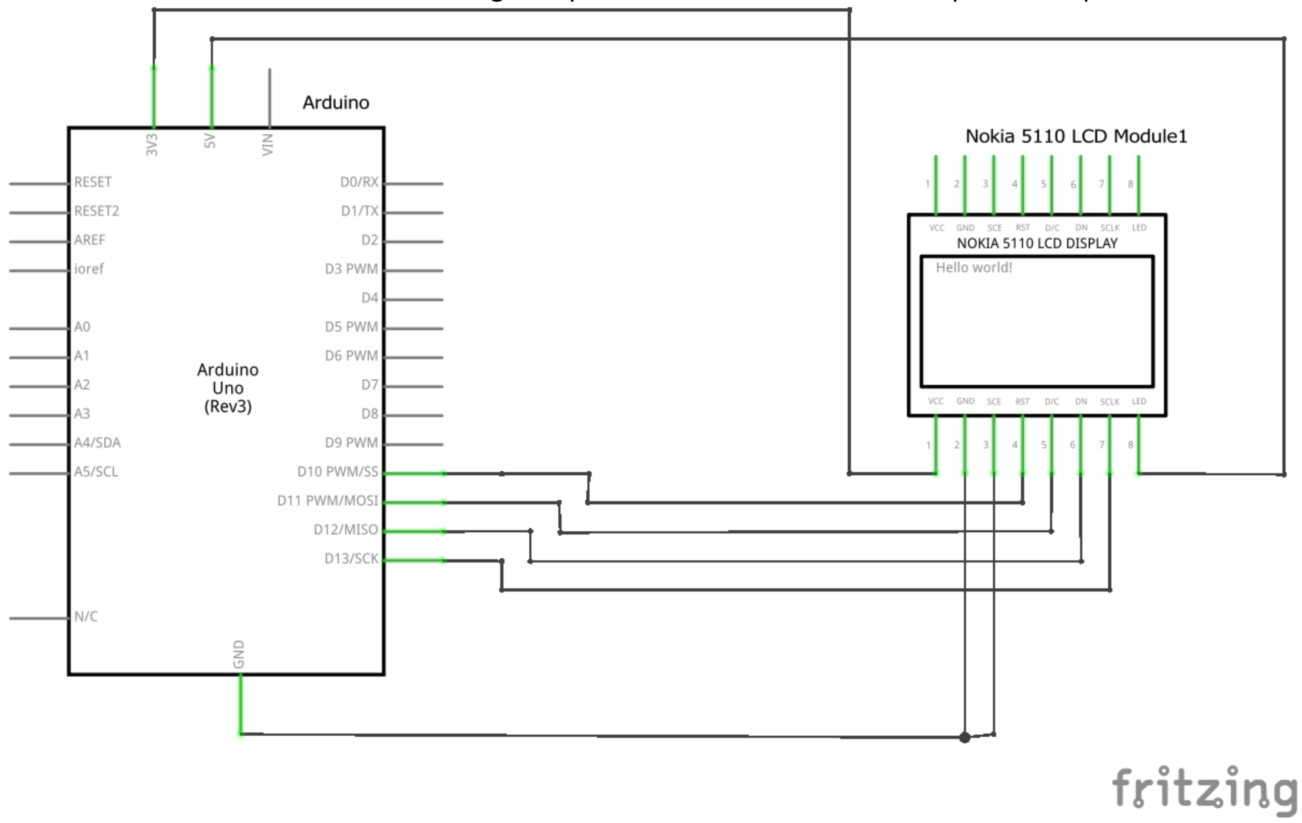
```
void readDHT() {
  if (state == RING)
    return;
  byte temp, hum;
  dht11.read(&temp, &hum, NULL);
  nokiaDisplay.setCursor(12, 24);
  nokiaDisplay.print("                             ");
  nokiaDisplay.setCursor(12, 24);
  nokiaDisplay.print(temp);
  nokiaDisplay.print("*C ");
  nokiaDisplay.print(hum);
  nokiaDisplay.print("%RH");
  nokiaDisplay.display();
}
```

The function reads the DHT by using the simpleDHT library's read() function. The received values are then printed to the Nokia 5110 display. We set the cursor at the appropriate position and print a series of spaces to delete the previous values. Then we print the new values and refresh the display. Notice that if the alarm is ringing we aren't refreshing the values: this isn't strictly necessary by itself but it is better to not poll the sensor while we are simultaneously running also the game and the music, to avoid unwanted latencies.

## Nokia 5110 display

This display was originally developed to be used with the iconic Nokia 5110 mobile phone. So it is capable of displaying alphanumeric characters, draw lines and other shapes and even displays a bitmap image. All this is possible because of its (84×48) monochrome pixels.[24]

The module comes with the PCD8544 interface IC which makes this module easy to use with low-level microcontrollers. It communicates through SPI protocol and hence does not require more pins.



[24] https://components101.com/nokia-5110-lcd

Libraries used to interact with the device are Adafruit_PCD8544.h[25] and Adafruit_GFX.h.[26]

Adafruit_PCD8544.h improve interaction with the display. It is responsible for operations like writing pixel values to the display memory.

Adafruit_GFX.h facilitate displaying simple polygons like lines and squares by its well optimize custom functions

The library supports both hardware SPI and software SPI, but in order to save pins we use software SPI and we tie the CS pin to ground.

The SPI communication happens with this function (register operations shown as comments):

```
 inline                              void
 Adafruit_PCD8544::spiWrite(uint8_t  d)
 {
      if (isHardwareSPI()) {
       // Hardware SPI write.
       SPI.transfer(d);
      }
      else {
        // Software SPI write with bit banging.
        for(uint8_t bit = 0x80; bit; bit >>= 1) {
          *clkport &= ~clkpinmask; // PORTB &= ~(1<<PORTB5)
           if(d & bit) *mosiport |=  mosipinmask; // PORTB |= (1<<PORTB4)
           else        *mosiport &= ~mosipinmask; // PORTB &= ~(1<<PORTB4)
          *clkport |=  clkpinmask; // PORTB |= (1<<PORTB5)
        }
      }
    }
```

In the code we insatiate new object responsible for communication with display

```
#define RST 10 - LCD reset

#define DIN 12 - Serial data out
#define DC 11  - Data/Command select
#define SCLK 13 - Serial Clock out

Adafruit_PCD8544 nokiaDisplay = Adafruit_PCD8544(SCLK, DIN, DC, RST);
```

[25] https://github.com/adafruit/Adafruit-PCD8544-Nokia-5110-LCD-library
[26] https://github.com/adafruit/Adafruit-GFX-Library

In set up functions system start the display then it sets the contrast for display. Next it assign text color, hence text is set to be black and the background is set to be white, then we clearing the display.

```
 nokiaDisplay.begin();
 nokiaDisplay.setContrast(30);
 nokiaDisplay.setTextColor(0xFFFF, 0x0000);
 nokiaDisplay.clearDisplay();
 nokiaDisplay.display();
```

Next function where we use functionality of the display is drawArrow function. In this case we also use defined in Adafruit_GFX.h functions that are responsible for drawing lines.
We defined to points that are beginning and end of the arrow and we specify the color of the particular arrow

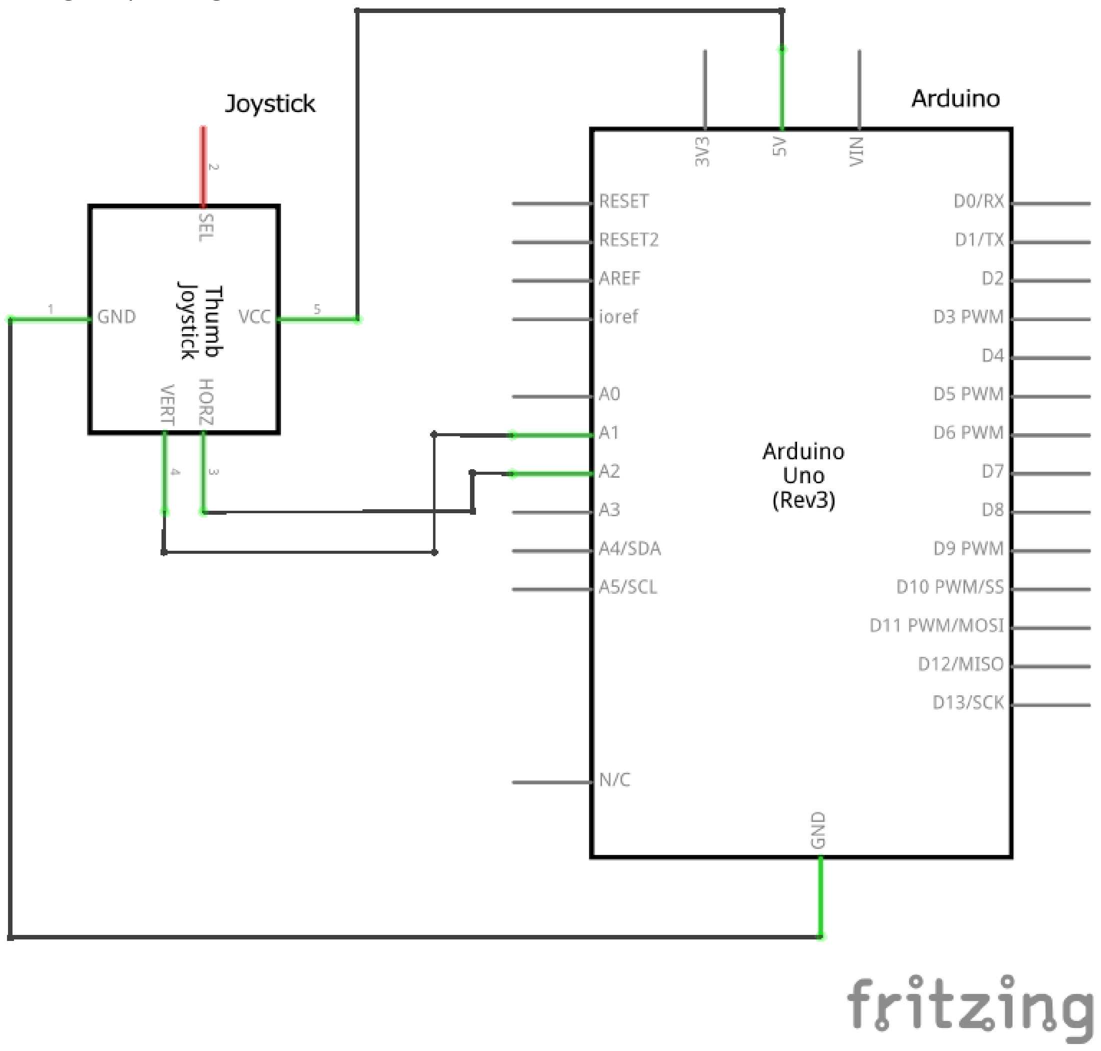
```
void drawArrow(int x, int y, int direction_, int size_, int color_) {
  byte color = 1;
  if (color_ == 0)
    color = 0xFFFF; //BLACK
  else
    color = 0x0000; //WHITE

  if (direction_ == 0 || direction_ == 1) {
    nokiaDisplay.writeLine(x, y - size_, x, y + size_, color);
    if (direction_ == 0) {
      nokiaDisplay.writeLine(x, y - size_, x + size_ , y, color);
      nokiaDisplay.writeLine(x, y - size_, x - size_, y, color);
    }
    else {
      nokiaDisplay.writeLine(x, y + size_, x + size_ , y, color);
      nokiaDisplay.writeLine(x, y + size_, x - size_, y, color);
    }
  }
  else {
    nokiaDisplay.writeLine(x - size_, y, x + size_, y, color);
    if (direction_ == 2){
      nokiaDisplay.writeLine(x - size_, y , x, y - size_, color);
      nokiaDisplay.writeLine(x - size_ , y , x , y + size_, color);
    }
    else{
      nokiaDisplay.writeLine(x + size_, y, x , y + size_, color);
      nokiaDisplay.writeLine(x + size_, y, x, y - size_, color);
    }
  }
  nokiaDisplay.display();
}
```

In this functions we use drawRect() that takes five arguments so first two is the position of left up corner of the rect second pair is the width and length of the rect and the last one in color of the rect (in this case 0xFFF is black).
When it finished drawing the frame of rect with fill it with black color.

```
void dotsDisplay(int numberDots) { //check this function
  for (int i = 0; i < numberDots; i++) {
    nokiaDisplay.drawRect(53 + 12 * i , 41, 3, 3, 0xFFF);
    nokiaDisplay.fillRect(53 + 12 * i , 41, 3, 3, 0xFFF);
  }
  for (int i = numberDots; i < 3; i++) {
    nokiaDisplay.drawRect(53 + 12 * i , 41, 3, 3, 0x0000);
    nokiaDisplay.fillRect(53 + 12 * i , 41, 3, 3, 0x0000);
  }
  nokiaDisplay.display();
}
```

## Joystick

This the device which as an output give the position of the stick accordingly to the OX and OY axis of the joystick. In fact it is nothing but two potentiometers that allow us to messure the movement of the stick in 2-D. Potentiometers are variable resistors and, in a way, they act as sensors providing us with a variable voltage depending on the rotation of the device around its shaft.



In setup function we define two pints A2 and A1 on which we read analog output of joystick.

```
#define OX A2

#define OY A1
```

In our program joystick is mainly use in the game which plays the role of the controller. To win the game user have to make sequence of proper movements on joystick.

`analogRead(OX) < 24` means that we make movement to the left
`analogRead(OX) > 1000` means that we make movement to the right

`analogRead(OY) < 24` means that we make movement to the bottom
`analogRead(OY) > 1000` means that we make movement to the top

Naturally the minimum value that system can obtain from the analogRead is 0 and the maximum value is 1024, but we want give player ability to make movements in range of acceptable error to make game more fun.

```
void changeOX() {
  if (arrows[currentArrow] == 2) {
    if (analogRead(OX) < 24) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }
  }
  if (arrows[currentArrow] == 3) {
    if (analogRead(OX) > 1000) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }
  }
}
void changeOY() {
  if (arrows[currentArrow] == 0) {
    if (analogRead(OY) > 1000) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }

  }
  if (arrows[currentArrow] == 1) {
    if (analogRead(OY) < 24) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }
  }
}
```

And the second place when the system use joystick is function responsible for checking if joystick is placed in the middle in range of acceptable error.

```
void zeroState() {
  if (analogRead(OX) < 600 && analogRead(OX) > 400 && analogRead(OY) < 600 &&
analogRead(OY) > 400) {
    comeBackToZero = true;
  }
}
```

To read the joystick values using the registers we have to use the Arduino ADC:

We set the pin from which we want to read in register ADMUX, writing 1 in MUX0 selects analog pin 1 while writing 1 in MUX1 selects analog pin 2. We use the former to read the Y-axis and the latter to read X-axis.

```
ADMUX = (1<<MUX0);
```
or
```
ADMUX = (1<<MUX1);
```

Then by writing 1 in position REFS0 we are using the reference voltage of 5V.

```
ADMUX |= (1 << REFS0);   // use AVcc as the reference
```

The ADCSRA is used firstly to set the proper prescaler, 128, by writing 1 in the 3 least significant positions. This prescaler gives the most accurate results, but each reading will take 104μs.
Then we set 1 in ADEN to enable the ADC circuit and we set 1 in ADSC to start the conversion.

```
// 128 prescale for 8Mhz
  ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
  ADCSRA |= (1 << ADEN);    // Enable the ADC
  ADCSRA |= (1 << ADSC);    // Start the ADC conversion
```

This loop is used to wait for the end of the conversion, by checking when the bit ADSC in ADCSRA is set to 1 by the MCU.

```
while(ADCSRA & (1 << ADSC)); // waits for the ADC to finish
```

The converted value is read from ADCL and ADCH register, which contain the least significant and most significant bits respectively. ADCH only contains 2 useful bits while the rest are reserved, so we are getting a 10bit return value.

```
int ADCval;
ADCval = ADCL;
ADCval = (ADCH << 8) + ADCval;
```

## The arrow game

This game needs analog joystick with two axis available and the screen. The goal of the game is two make sequence of movements, that are represented by directional arrows on the displays, in defined time.

On start, game function draw the arena for the game. Next system generate sequence of random integers in range from 0 to 3. Each digit represent a different direction 0 - up 1 - down 2 - left 3 - right.

```
byte arrows[HOWMANY];

byte currentArrow = 0; //which arrow is processed right now
byte numberDots = 3; //how many dots there gonna be
unsigned long timeChecker; //timer
bool comeBackToZero = true; //after every good answer you have to come back to 0
position
bool winTheGame  = false; // global


void giveValues() {
  for (int i = 0 ; i < HOWMANY; i++){
    arrows[i] = random(0, 4); // four signs <- -> ^ .
  }
}

void printArena() {
  for (int temp_y = 10; temp_y < 48; temp_y += 37) {
    for (int temp_x = 0; temp_x < 84; temp_x++)
    {
      drawAPixel(temp_x, temp_y);
      if (temp_y == 0)
        drawAPixel(temp_x, temp_y + 1);
      else
        drawAPixel(temp_x, temp_y - 1);
    }
  }
  for (int temp_x_ = 0; temp_x_ < 84; temp_x_ += 83) {
    for (int temp_y_ = 10; temp_y_ < 48; temp_y_++)
    {
      drawAPixel(temp_x_, temp_y_);
      if (temp_x_ == 0)
        drawAPixel(temp_x_ + 1, temp_y_);
      else
        drawAPixel(temp_x_ - 1, temp_y_);
    }
  }
  nokiaDisplay.display();
}
```

System shows sequence on the screen and in the left down corner it displays the text that describe current arrow direction.

```
void getCurrentArrow() {
  if ( arrows[currentArrow] == 0) {
    printText(" UP   ", 2, 40, 1);
  }
  if ( arrows[currentArrow] == 1) {
    printText(" DOWN ", 2, 40, 1);
  }
  if ( arrows[currentArrow] == 2) {
    printText(" LEFT ", 2, 40, 1);
  }
  if ( arrows[currentArrow] == 3) {
    printText(" RIGHT ", 2, 40, 1);
  }
}
}
```

If the user do the proper movement on joystick the arrow will disappear and text will change accordingly to the direction of the next arrow.

This is function responsible for evaluating correctness of the movement made by player:

```
void changeOX() {
  if (arrows[currentArrow] == 2) {
    if (analogRead(OX) < 24) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }
  }
  if (arrows[currentArrow] == 3) {
    if (analogRead(OX) > 1000) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }
  }
}
void changeOY() {
  if (arrows[currentArrow] == 0) {
    if (analogRead(OY) > 1000) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }

  }
  if (arrows[currentArrow] == 1) {
    if (analogRead(OY) < 24) {
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
      clearArrow();
      currentArrow++;
      if (currentArrow < 4)
        getCurrentArrow();
      comeBackToZero = false;
    }
  }
}
```

To prevent the situation when player just make circular movements on analog stick and win the game we introduce the flag that checks, if after every well done movement, player come back to zero position of the joystick according to the x and y axis.

```
void zeroState() {
  if (analogRead(OX) < 600 && analogRead(OX) > 400 && analogRead(OY) < 600 && analogRead(OY) > 400) {
    comeBackToZero = true;
  }
}
```

In the state of the game when current arrow is second arrow game activate the dots system. On start up of the game, system display three dots in the right down corner of the screen. After defined amount of time if user wan't do the proper movement first dot will disappear and then after the same amount of time the second and so on.

```
void changeNumberDots() {
  if (currentArrow > 0){
    if (numberDots <= 0) {
      currentArrow--;
      getCurrentArrow();
      redrawArrow();
      timeChecker = millis();
      numberDots = 3;
      dotsDisplay(numberDots);
    }
  }
}
```

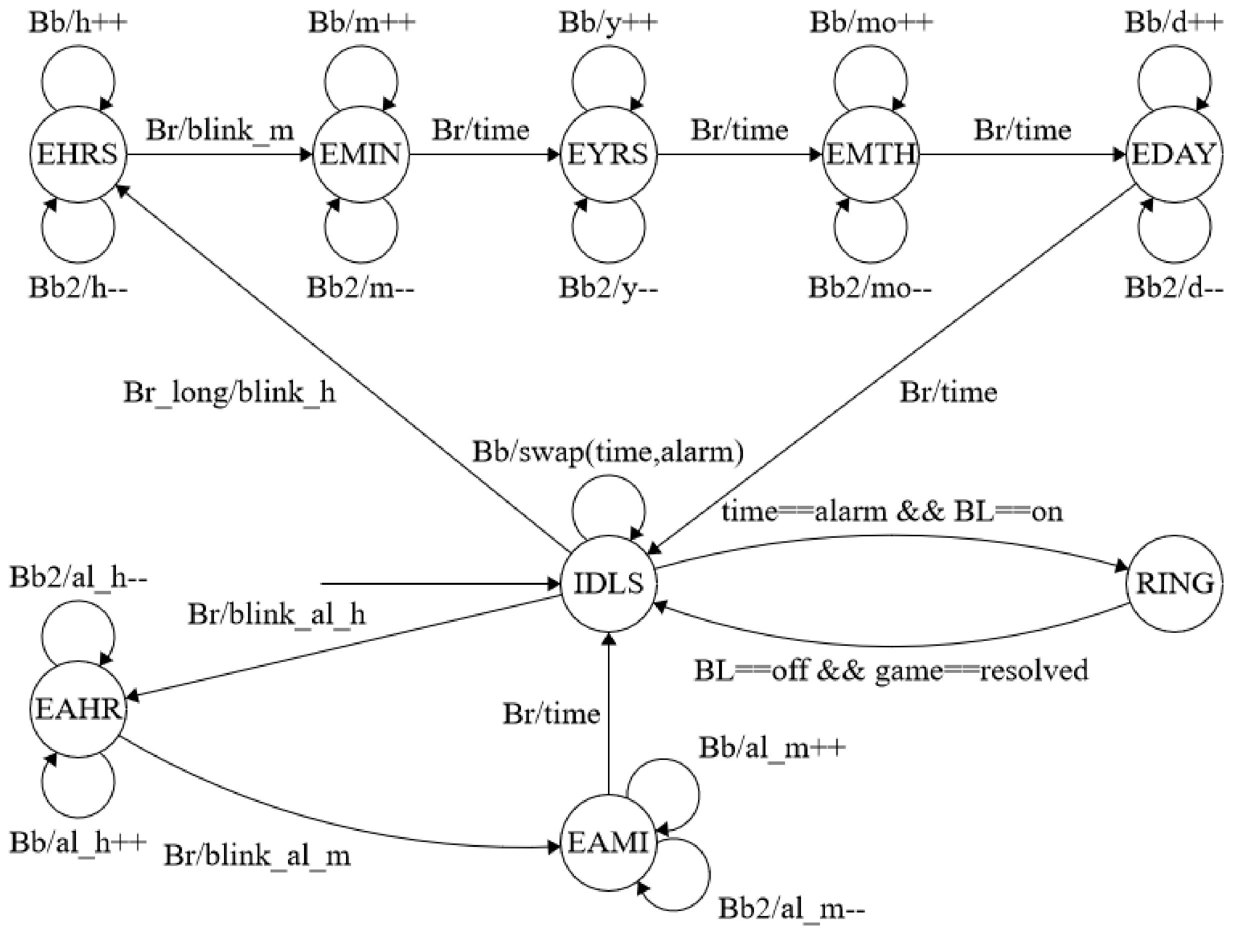
If player won't make on time the proper movement and the last dot disappear, the system will show again the last hidden arrow on the screen and the text will change accordingly and as a result user will have to redo last movement. Naturally the "dot system" does not work in case of the first arrow because we don't have the place to come back.

The game is stepped by this function, which is run in the loop() if we are in RING state:

```
void runGame()
{
  if (winTheGame == false) {
    if (comeBackToZero == true) {
      changeOX();
      changeOY();
    }
    zeroState();
    changeNumberDots();
    if (!checkWin()) {
      if (millis() - timeChecker > TIMEFORGOODANSWER) {
        if (numberDots > 0 && currentArrow > 0){
          numberDots--;
          dotsDisplay(numberDots);
        }
        timeChecker = millis();
      }
    }
  }
}
```

If the game isn't won and the joystick was in zero state last time, then we read the new jostick positions and eventually the functions also step the game ahead in case of a correct move. Then we check if we have to reset the number of dots back to 3 by calling chanegNumberDots(). Finally we check for winning condition, and if it’s false, we check if the time elapsed for the move is finished, if yes we decrease the number of dots.

## Finite state machine



The finite state machine above describes the behaviour of the system.

The possible states are:

- IDLS: idle state
- EAHR: editing alarm hours
- EAMI: editing alarm minutes
- RING: the buzzer is ringing
- EHRS: editing current time hours
- EMIN: editing current time minutes
- EYRS: editing date year
- EMTH: editing month
- EDAY: editing day

At the beginning, the default state is IDLS (idle). In this state the 7-segment displays the current time, and the colon between hours and minutes blinks. Pressing the first black button, the display view will switch to alarm. In this view the colon does not blink.

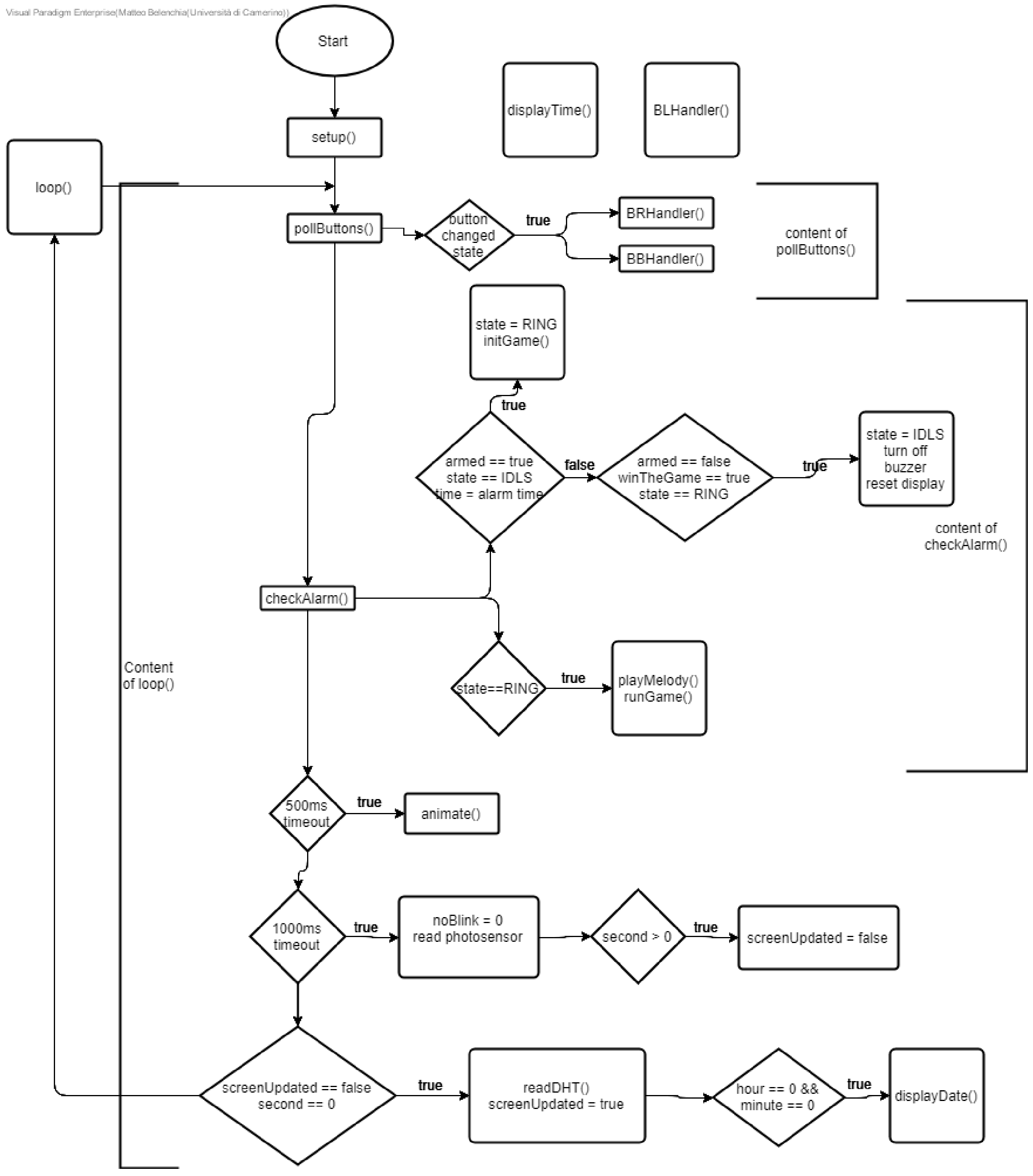
Starting from the IDLS state, after pressing the red button the system switches on EAHR state: edit alarm hours. A press on the first black button increases alarm hours, a press on the second black button decreases it and a press on the red button switches to the EAMI state, to edit alarm minutes. As before, it is possible to increase and decrease values by pressing black buttons. Another press on the red button and the system goes back to IDLS state.

Another transition from IDLS state is possible by pressing for a long time the red button: it makes the state switches to EHRS (editing hours). By pressing the first black button is possible to increase hours, while the second black button will decrease it. At this point, a short press on the red button and is possible to edit minutes (EMIN), with the same method: first black button to increase, second to decrease by one.
In these states the display shows the time modified blinking respectively hours and minutes. From the EMIN state, pressing on the red button switches the system to the editing date mode: first EYRS to modify current year, then EMTH to modify current month, then EDAY to modify current day. In each of these states, a press on the first black button increases value by one, while a press on the second black button decreases it. While editing date, the 7-segments displays current time, while the date is shown on the Nokia display, without blinking.
After editing date, it is possible to go back to IDLS state with another press on the red button.

Last transition occurs during alarm: when alarm and current time are the same, and the alarm switch is ON, the system commutes to RING state, where the buzzer starts playing. In order to go back to idle, the alarm switch must be turned OFF and the game must be resolved.

## Algorithm outline

In this section we outline the algorithm from a high level.



After the setup function is executed, the flow chart shows a dissection of the loop() function. The loop() function roughly executes 2 functions and executes certain instructions if enough time has elapsed from a certain starting point. On the right side of the chart there is a dissection of the instructions of these 2 functions.

The first function is pollButtons(), which checks if a button has changed state (with debouncing), and if true it will call the appropriate handler passing the new state as argument.

The second function is checkAlarm() and makes some checks related to the RING state, that is the state when the alarm is triggered. In the most basic situation, when the clock has not armed an alarm, nothing is executed by the function. In this case the function runs in 4 to 8 $\mu s$. Then, if the alarm is armed, the clock is in IDLS state and it is the time set for the alarm, the first left-most check evaluates to true and it initializes the game and sets the state to RING. The right-most check still doesn't evaluate to true. The bottom check though is now execute and runs playMelody() and runGame(). Now at each call the function runs in 572 to 576 $\mu s$. The former runs the alarm sound, the latter runs the game. At some point the game will be won and the flag winTheGame is set to true, and also the user will have toggled the DPDT switch which will set the armed flag to false. The state was previously set to RING, and no button will ever change RING to any other state, so the state is still unchanged. At this point, executing checkAlarm(), the right-most check evaluates to true and it will set the state back to IDLS, turn off the sound and clear the Nokia display where the game had run, all of this in about 1ms.

Later, 3 time related checks are evaluated.

The first checks if the timeout for the animations has exceeded 500ms. When it has, it will either display the next image in the animation or put the LED matrix in power saving mode if the animation has been repeated 3 times already since the last call to startAnimation(). The variable keeping the time is refreshed anytime an image has been displayed.

The second check represents 2 different if clauses, both with timeout 1000ms. The first one resets the noBlink flag after the blinkTimer had been refreshed on the last of the black buttons presses. The second one reads the analog value from the photosensor and changes the brightness of the 7-segment displays and the LED matrix. The secondPeriod variable is refreshed everytime this if clause is entered, so it will actually execute about every 1 second. Also, if the current second (of the time kept by the RTC clock) is greater than 0, is sets the screenUpdated flag back to false.

The third check uses the RTC time to check if the current second is 0, or in other words if the RTC is at the start of the new minute, and if screenUpdated is false, i. e. one real minute since the last refresh has elapsed. The variable second is refreshed at each loop() call so the contents of this check are executed every one minute. When the check evaluates to true we refresh the date if also the current minute and hour is 0, while we refresh temperature and humidity regardless.This is achieved by calling displayDate() and readDHT().Then we set screenUpdated to true because we don't want to run this code again while the current second is still 0.

The 2 process boxes disconnected from the flowchart represent interrupts. The first interrupt routine, displayTime(), executes every 500ms and it is used to display the current time, alarm time and eventually also blink what it is being shown. It runs in 1016 to 1020 $\mu s$. The second interrupt routine runs only when there's a change in the level in pin 2. When run, it takes 8 $\mu s$.

## Failure mode, effects and criticality analysis

The project features many different devices upon which it depends, with different degrees of importance. We applied here the FMECA analysis with the relevant categories of Probability, Severity and Risk Level which is a relation of the previous two factors by the formula:

$$Risk\ Level = Probability\ \times Severity$$

The Risk column shows the name of the evaluated risk. For the simplest items we evaluated a cable disconnection rather than break down because we deemed such an extremely unlikely occurrence for them to break down that it would be just a waste of space to describe it. The Probability column shows the likelihood for the event to occur according to this table[27], while the Severity shows the gravity of the effect when the event happens[28]. The Detectability column determines how the failure is detected and also includes a description of the effects that are experienced[29]. The Risk Level column evaluates the above formula[30]. The Reaction column shows the steps to take in order to solve the issue.

| Risk | Probability | Severity | Detectability | Risk Level | Reaction |
|---|---|---|---|---|---|
| **Burned board** | Remote(B) | Catastrophic (5) | Certain(1) There is no functioning at all. | High | Use a new board |
| **Degraded flash memory** | Extremely Unlikely(A) | Catastrophic(5) | Low(5) The board won't work at all but every other component would be just fine. | Moderate | Use a new board |
| **Power failure** | Reasonably Possible(D) | Catastrophic(5) | Certain(1) Everything is turned off. | Unacceptable | Connect the power supply properly. |
| **RTC disconnection** | Remote(B) | Catastrophic (5) | Almost Certain(2) The screens freeze and buttons don't respond. | High | Reconnect the RTC and reset the board |
| **DHT sampling error** | Occasional(C) | Very Minor(2) | High(3) The temperature reading might appear to be obviously wrong and the system is less responsive. | Moderate | Reconnect the DHT properly |
| **Buzzer disconnection** | Remote(B) | Critical(4) | High(3) The alarm makes no noise. | Moderate | Reconnect the Buzzer properly |
| **7-segment display failure** | Extremely Unlikely(A) | Critical(4) | Certain(1) The display is turned off and doesn't show time. | Low | Replace the display |
| **LED matrix failure** | Extremely Unlikely(A) | Minor(3) | High(3) The LED matrix doesn't light up upon red | Low | Replace the LED matrix |

[27] https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis#Probability_(P)
[28] https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis#Severity_(S)
[29] https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis#Detection_(D)
[30] https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis#Risk_level_(P*S)_and_(D)

| | | | | | |
|---|---|---|---|---|---|
| | | | button presses or alarm triggering. | | |
| **Disconnected black buttons** | Occasional(C) | Critical(4) | Moderate(4) The buttons are unresponsive, but it is only detected when using them. | Moderate | Reconnect the button(s) |
| **Disconnected red button** | Remote(B) | Critical(4) | Moderate(4) The button is unresponsive, but is only detected when using it. | Moderate | Reconnect the button |
| **Erratic red button behaviour** | Reasonably Possible(D) | Minor(3) | High(3) It can be noticed that pushing the buttons leads to incorrect behaviour in the form of skipping machine states, e.g going from editing hours straight to editing months. | Moderate | Increase the debouncing time or replace the button |
| **Disconnected DPDT switch** | Remote(B) | Critical(4) | Moderate(4) Alarm won't turn on or shut off, but it is noticed only when using it. | Moderate | Reconnect the button |
| **Disconnected potentiometer** | Remote(B) | Critical(4) | Moderate(4) The alarm won't sound at all, because the power to the buzzer goes through the potentiometer. | Moderate | Reconnect the potentiometer |
| **Disconnected photoresistor** | Remote(B) | Very minor(2) | Low(5) The brightness won't adjust at all, but the effect is in general slight. | Low | Reconnect the photoresistor |

| | | | | | |
|---|---|---|---|---|---|
| **Joystick failure** | Extremely Unlikely(A) | Critical(4) | Moderate(4) The game cannot be played. | Low | Replace the joystick |
| **Nokia display failure** | Extremely Unlikely(A) | Critical(4) | Certain(1) The display is turned off and doesn't show its data. | Low | Replace the display |

The board might experience 3 types of risk. The board might get completely broken (burned) if by human error the wrong voltage is supplied to the VIN pin. This has occurred to one of us in the past, so the risk is considered Remote. The other risk is that the memory might not work anymore after a certain number of writes, which is about 100.000. In both these cases the board might as well be considered dead and has to be substituted. A very common issue is a power failure, due to what seems to be a very sensitive USB connector. The USB cable must make a good contact for the board to work properly, otherwise everything is turned off.

If the RTC clock is disconnected the system freezes, likely when trying to read from it on the I2C interface. The soldering of the RTC isn't particularly good although it has never actually happened to disconnect if not for testing purposes.

Sometimes it occurred for the DHT temperature and humidity sensor to have issues, namely a failure in the reading. In those cases the temperature and humidity readings look wrong and the system slows down when executing the instructions to read the DHT, which take more time.

In case of a buzzer disconnection the alarm won't sound and it would make the whole system lose the functionality of being an alarm.

If the 7-segment display breaks down the system would become rather useless until substituted, because it won't show hours and minutes anymore. The LED matrix is less important and the system can work without it, but it is such an evident fault that it's nevertheless at least of "Minor" Severity.

In case of button disconnection the effect is that those buttons won't work and the related functionalities won't go. A special case of button problem is the behaviour of the red button which has shown to be quite erratic. The contacts seem to not be working too well and so it causes to give a series of HIGH and LOW signals to the MCU which might exceed the debounce time and cause multiple button presses. It can alleviated by increasing the debounce time.

A disconnection of the potentiometer would be equivalent to a disconnected buzzer because voltage passes through it.

A disconnection of the photoresistor wouldn't cause any trouble at all and it is of low importance.

Finally, a joystick or Nokia display problem would make playing the game impossible, with the added effect of not being able to see the date, temperature and humidity in the latter case.

## Limits and future advancements

The project has been completed in all its functionalities and maybe more, but some limitations have been noted.

A concern among some group members was about the limitedness of the buzzer's output. An idea of ours was to play some classical music over it, but being a simple piezoelectric buzzer, it cannot perform more than one sound at a time. Another limit was given by the led matrix, which is of a boring monochromatic red and only allows 8bit animations. The videogame display is also noticeably outdated (in fact, original Nokia 5110s were produced in 1998) and the temperature sensor is a really cheap one. Of course, the reasoning behind all of this is that we bought all the components using our funds and we didn't deem necessary to buy anything more than the bare minimum.

A somewhat harder limit is given by the number of pins on the MCU. It would have been good to add more buttons or more buzzers (again, to play more complex music), but we are already using all the available pins (except for 2 analog pins which we don't use to make the project strictly compatible with Arduino UNO). A possible solution would be using a more powerful MCU (Arduino Mega is an example) or buying an extension shield.

Another potentiometer could have been put to regolate the Nokia display backlight.

What we couldn't make up for the hardware could have been made with the software. Possible future expansions may feature:

- Setting of multiple alarms
- Toggle temperature from Celsius to Fahrenheit
- Setting of a different timezone
- Switching from 24h to 12h
- Year, Month and Day blinking when editing
- More than one melody