# Buttons

The system involves three different types of buttons.

The black button (BB) 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 increases and one decreases values by one.

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.

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 state and if more than 25 milliseconds passed.

## 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);

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 (direction == -1 && \*toEdit == 0){

   \*toEdit = mod;

 }

 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;

   }

 }

 \*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

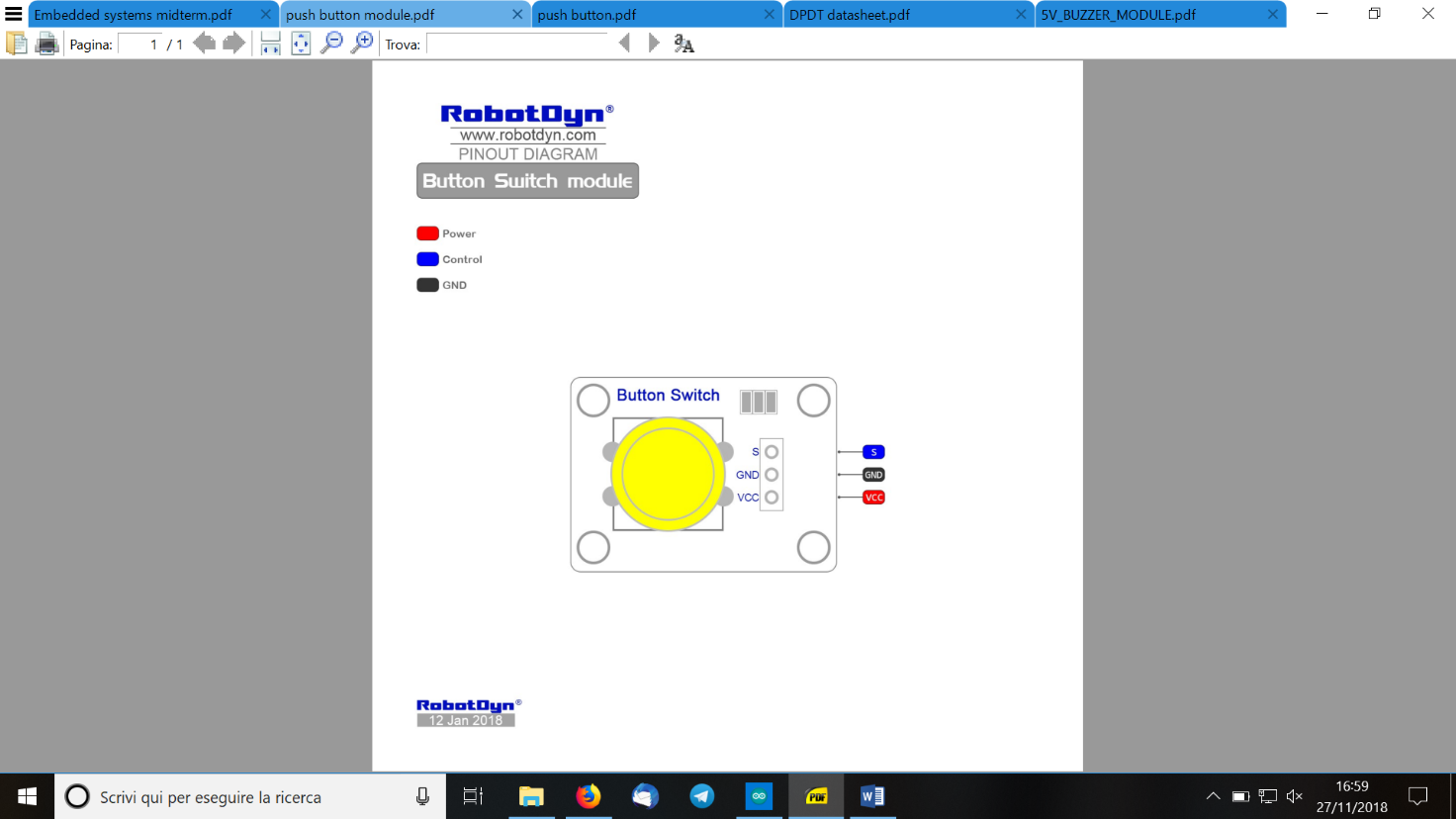
 displayTime();                  // display the new values immediately, not waiting till the next interrupt

 displayDate();

}

If the system is in the IDLS or RING state, pressing the button 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. At this point the value is edited, increased or decreased by one unit according to the pressed button. 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

And in the setup:

 pinMode(BR, INPUT);

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

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

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.