SCOREBOARD PROJECT

Electronic board
with 7-segments displays
and Arduino

version 1.0.2

Gionata Boccalini

February 10, 2013

Contents

List of Figures

Listings

1 Introduction

This document describes the design and realization of an electronic scoreboard, that I was asked to make for my local basketball team. I had some experience with 7-segments displays (from school) and with microelectronics so I decided to make my own scoreboard, using basic electronic blocks and Arduino as CPU. I had to seek for the right components on the Internet for months, and to put everything together in my garage, with the help of my father and some of my friends. Some parts were already made and some I had to build from scratch using tools that I would have never thought I used.The whole process took months but the final result was really impressive for me, given the knowledge and possibilities I had.

The rest of the document is organized in two main section, one describing the hardware part and the other speaking about the software. I will try to describe the most important aspects, using pictures, and give the main idea of simplicity and robustness that I tried to achieve.

I have to cite the source [1] where I found the inspiration to build this scoreboard, and from which I took some design hints. The site is run by Mr. Kianoosh Salami, who was really helping me in the process of finding the components and building the board.

I have also to thank many of my friends for the help, the knowledge the gave me and the inspiration derived from the discovery of the "Do it yourself"electronic field, represented by the Italian Arduino platform.

Thank you!!

2 Hardware

The scoreboard has been completely realized by hand, starting from scratch, using wood for the chassis to create a box with an open side where I could put the electronic components. We made some holes for the 7-segments displays and the external connections. Some pictures will explain better than words:

The chassis was robust enough to hold the displays, Arduino, and the wiring. The list of components, with pictures, is shown in section 2.1.

2.1 Components

The main components are:

- 7-segments displays: I found those ones in a Chinese electronic company site, Sure Electronics [2], selling worldwide, and I was very lucky because those displays already integrates the electronic logic to control the data flow from the CPU to the integrated shift registers. Thus I saved the effort to make my own circuit to drive the displays!!

[1] http://kiantech.me/entries/2011/6/21/live-nba-scoreboard.html
[2] http://www.sureelectronics.net/goods.php?id=721

Figure 1: A 7 inch 7-segments displays

- Arduino: this is an Italian prototype platform really easy to use, with open source software and standard hardware which helps non professional electronic designer and engineer to develop their own application. I used this CPU to control the whole panel, the displays and the digital inputs.

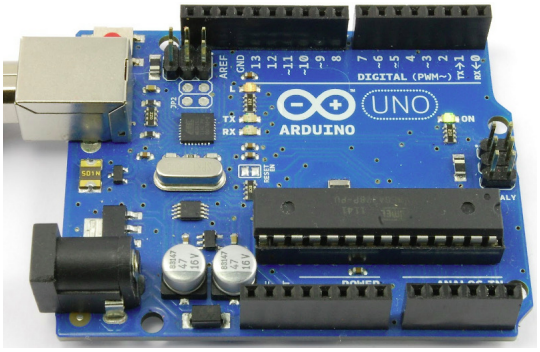

Figure 2: Arduino UNO platform

- homemade connection boards: I used those ones to connect the CPU to the rest of the circuit, like the control panel and the 7-segments displays. Later in this document or in the download section of the project site [3] you can find the Eagle schematic and board files.

[3] http://code.google.com/p/display-group/

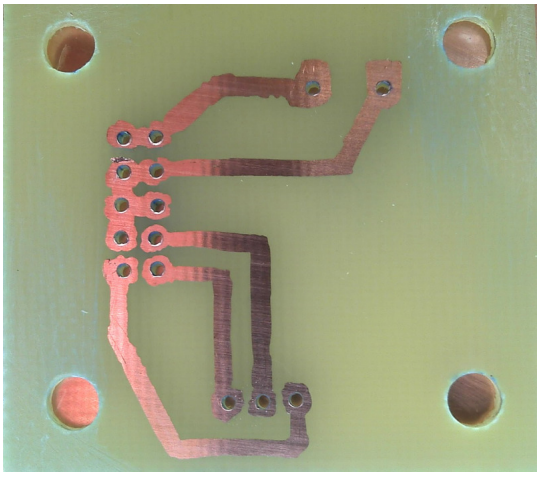

Figure 3: Homemade connection board without components

- wood and DIY materials.

2.2 ScoreBoard electrical connections

The 7-segments are powered by a 12 V PSU, I decided to connect the power in parallel to all the boards: the data link can carry 1 A current, but that is not enough to power 7 displays boards on the same bus. Considering that Arduino is also powered by the same PSU, and the input are also, I used a 6.75 A, 12 V PSU, just to be sure the current would be enough for everything. The figure 4 shows the back of the scoreboard with all the components:

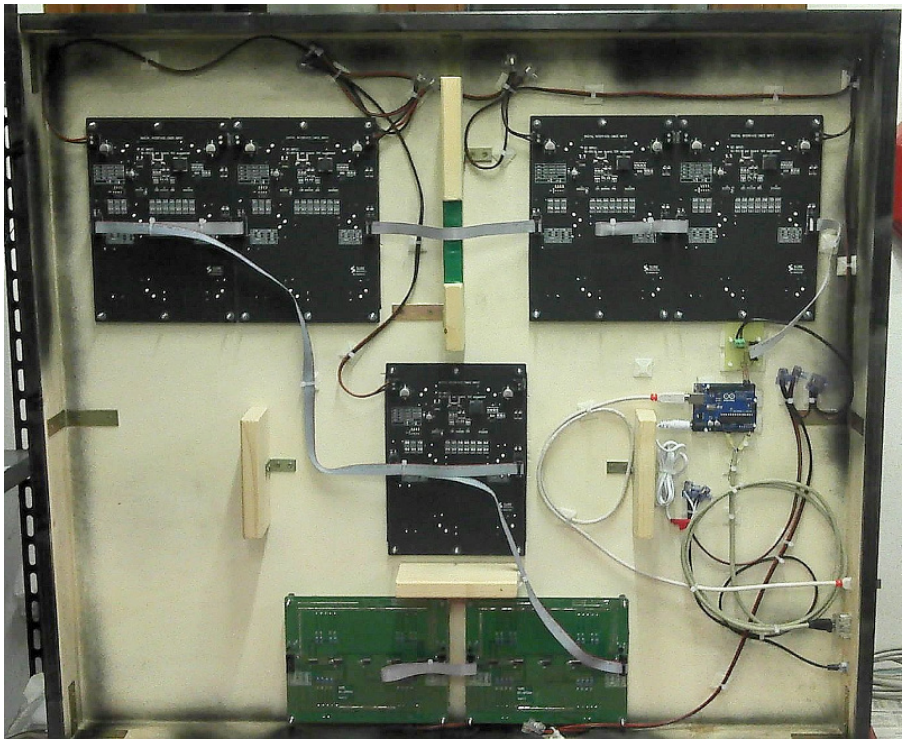

Figure 4: Back side of the board, with Arduino and electrical connections

You can also see the external connections, in the bottom right corner:

- Power cord: to power the whole board;
- USB: to program the Arduino when the board will be installed on the wall;
- Ethernet: to connect the control panel.

The data link between the CPU and the displays is made with a 10 pole IDC cable, with three data pins, as shown in figure 5.

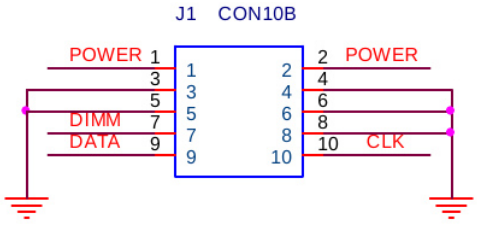

Figure 5: Power and data connection on the 7-segments board

The pins are described as follows

- **DIMM/OE**: to control the brightness of the LEDs and the output enable of the shift registers;
- **DATA**: to shift in data bits;
- **CLK**: to clock the data bits on the bus.

To have more information on the displays and the electronic behind look on the project site, download section, for the displays datasheets.
I used an electronic board to make the data connection persistent, as shown in figure 3, with standard connectors soldered on it: figure 6 shows the board PCB circuit made with Eagle CAD software.

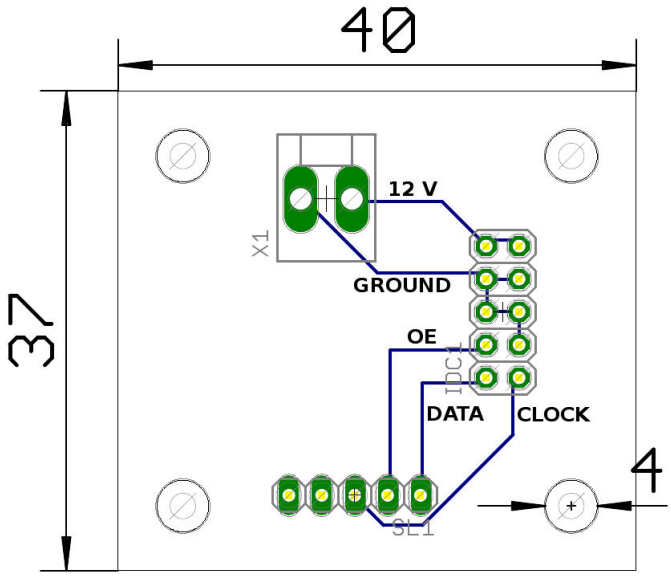

Figure 6: Connection board schematic

The OE, DATA, CLOCK pins come from Arduino, and are routed to the IDC connector, together with the 12 V line and ground. The IDC cable goes to the first 7-segments display (top right

corner in figure 4) the IDC output connector goes to the second display on its left, and so on. Using a digital shift out technique on Arduino it is possible to update all the displays at a time. The DisplayGroup library has been used for this purpose.

The two displays at the bottom of figure 4 are the last two of the queue: they are 4 inch tall and use the same hardware interface of the 7 inch displays, but the segments bit code is different! The DisplayGroup library can handle this representation difference in transparent way to the user.

2.3 Control panel

The small control panel shown in figure 7 has been built to give the user the possibility to remotely control the scoreboard during the game. The buttons are described in picture 8.

Figure 7: Control panel

There is a "shift"like button to enable additional function like the set of the timer and the reset of score.

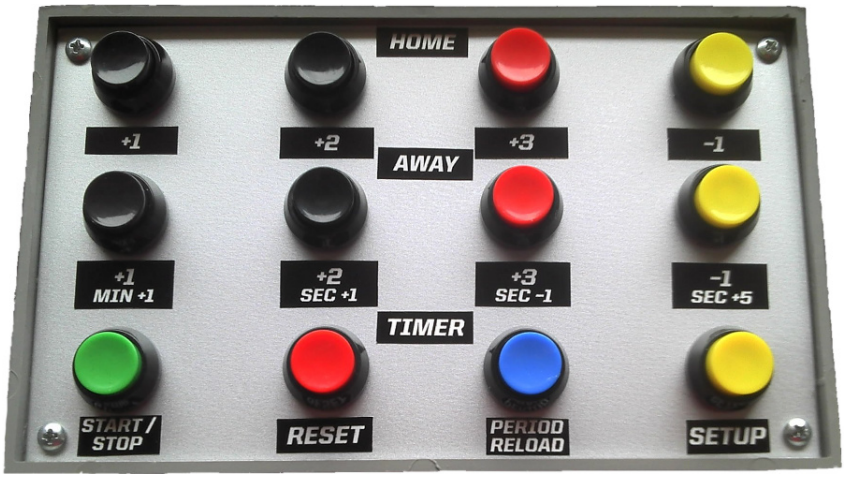

Figure 8: Control panel front view

The panel is powered by the Arduino 5 V output through the Ethernet cable; this is the only cable needed to make the panel work: it brings the power and the data link, using only five cable out of the eight present in the Ethernet cable. Those cables are mapped to the Arduino analog input: this means that, since the buttons are digital switch, I had to build a board to extend the Arduino input capability: using only 3 analog input Arduino can receive 12 digital inputs, with the correct resistors and the help of the software [4]. The board schematic is shown in figure 9:

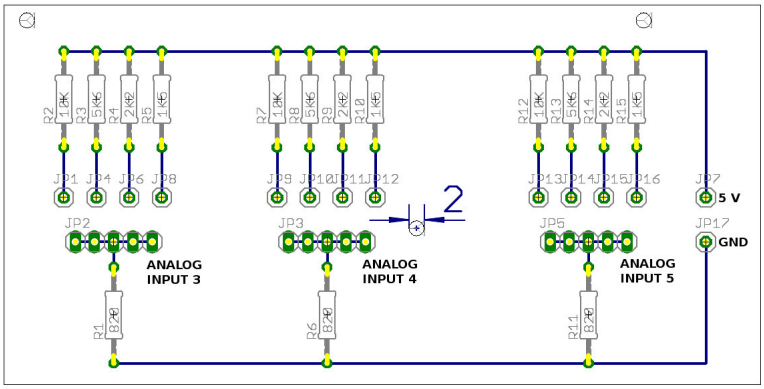

Figure 9: Control panel connection board

The board is essentially a voltage divider with three voltage sample, one for each analog input: every button has its particular size of the divider resistor, so when the user pushes a button the corresponding voltage value goes to the correct analog input. Those values are hard coded into the control program on the CPU; the library for the input control scans the analogs input and calls the correct function when needed. This library will be described in section 3.1.

I used a straight Ethernet cable but, due to the necessary joints, I had to map the cable differently between the two communication sides: as you can see from picture 10 the orange-white e green-white cables are swapped:

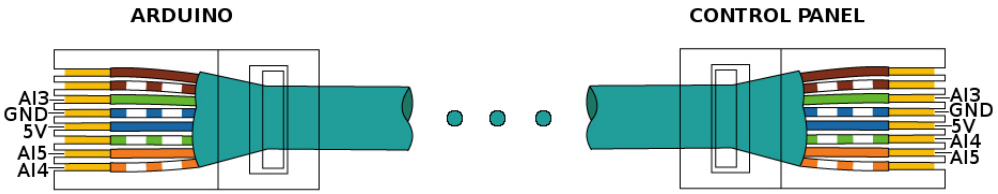

Figure 10: Control panel connection board

By using this cabled solution I was able to control the scoreboard with a quite long cable (12 meters), while keeping the realization cost low. The other solution would be a wireless communication, but that would also be more expensive.

[4] http://www.instructables.com/id/How-to-access-5-buttons-through-1-Arduino-input/?ALLSTEPS

3 Software

The control software is made of three library and a main logic, written in C++, with the use of Arduino API and C++ STL library (Arduino version). The README and INSTALL document in the root folder of the project explain where to find those libraries, and then you can understand how to compile them. The rest of the compilation process is done by Eclipse or by the makefile. Also the flashing of the program on the Arduino program memory is automatically done, provided that all the needed tools are installed.

3.1 Dependences

The most important dependence is the DisplayGroup library, which has already been described in the documentation at the project folder, on the Google code svn server (INSTALL file). This library is needed to support in an efficient way many 7-segments connected to a serial bus, each with a shift register, just like the situation in this project as shown in figure 4. The display configuration in this case is:

Listing 1: DisplayGroup configuration

```
1 DisplayGroup::DisplayManager disManager(PIN_COM_DATA, PIN_COM_CLOCK,
      PIN_OUTPUT_ENABLE);
2
3 disManager.addGroup("Home_score", 2, 0, &score.home, gDigits7, sizeof(gDigits7));
4 disManager.addGroup("Away_score", 2, 1, &score.away, gDigits7, sizeof(gDigits7));
5 disManager.addGroup("Period", 1, 2, &time.period, gDigits7, sizeof(gDigits7));
6 disManager.addGroup("Timer_minutes", 2, 3, &time.min, gDigits4, sizeof(gDigits4));
7 disManager.addGroup("Timer_seconds", 2, 4, &time.sec, gDigits4, sizeof(gDigits4));
```

where `score` and `time` are struct I used to keep information regarding the match score and the actual time. `gDigits7` and `gDigit4` are arrays of segments code, necessary for the library configuration, respectively for 7 inches and 4 inches displays.

The second dependence library is AnalogButtons [5], used to control many digital inputs through few analog inputs from the Arduino interface. The configuration for this library is:

Listing 2: AnalogButtons configuration

```
1 // Patch cable
2 const uint16_t bAVal[10] = { 60, 90, 120, 150, 250, 295, 340, 375 };
3
4
5 AnalogButtons homeButtons(HOME_ANALOG_INPUT, DEBOUNCING_COUNT_HOME, &
      handleHomeButtons);
6 Button b1 = Button(HOME_P1, bAVal[0], bAVal[1]);
7 Button b2 = Button(HOME_P2, bAVal[2], bAVal[3]);
8 Button b3 = Button(HOME_P3, bAVal[4], bAVal[5]);
9 Button b4 = Button(HOME_M1, bAVal[6], bAVal[7]);
10
11 AnalogButtons awayButtons(AWAY_ANALOG_INPUT, DEBOUNCING_COUNT, &handleAwayButtons)
      ;
12 Button b5 = Button(AWAY_P1, bAVal[0], bAVal[1]);
13 Button b6 = Button(AWAY_P2, bAVal[2], bAVal[3]);
14 Button b7 = Button(AWAY_P3, bAVal[4], bAVal[5]);
15 Button b8 = Button(AWAY_M1, bAVal[6], bAVal[7]);
16
17 AnalogButtons timerButtons(TIMER_ANALOG_INPUT, DEBOUNCING_COUNT, &
      handleTimerButtons);
18 Button b9 = Button(TIMER_START_STOP, bAVal[0], bAVal[1]);
19 Button b10 = Button(TIMER_RESET, bAVal[2], bAVal[3], HELD_DURATION);
20 Button b11 = Button(PERIOD_P1, bAVal[4], bAVal[5]);
21 Button b12 = Button(SETUP_MODE, bAVal[6], bAVal[7], HELD_DURATION);
22
23 ...
24
```

[5] http://playground.arduino.cc/Code/AnalogButtons

```
25 // Input Buttons
26 homeButtons.addButton(b2);
27 homeButtons.addButton(b1);
28 homeButtons.addButton(b3);
29 homeButtons.addButton(b4);
30
31 awayButtons.addButton(b5);
32 awayButtons.addButton(b6);
33 awayButtons.addButton(b7);
34 awayButtons.addButton(b8);
35
36 timerButtons.addButton(b9);
37 timerButtons.addButton(b10);
38 timerButtons.addButton(b11);
39 timerButtons.addButton(b12);
```

I used three analog inputs to control twelve digital inputs, called buttons in the program. For each analog input there is a callback function: `handleHomeButtons`, `handleAwayButtons` and `handleTimerButtons`. In these functions is implemented the control logic for each buttons. Some of the buttons have an `HELD_DURATION` options: these inputs must be held for some seconds to become active.

3.2 Main control program

The main program is composed by the three library described in section 3.1 and by a main program that will be described in this section. The structure of the main program is shown in 3 there are three callbacks to handle the analog input, one to handle the timer interrupt, and some functions to manage the EEPROM memory.

Listing 3: Main program structure

```
1
2 void handleHomeButtons(int id, boolean held) {}
3
4 void handleAwayButtons(int id, boolean held) {}
5
6 void handleTimerButtons(int id, boolean held) {}
7
8 void handleTime() {}
9
10 // ##########################################################
11 // ####### EEPROM wear leveling algorithm functions ########
12 // ##########################################################
13
14 void resetEEPROM() {}
15
16 boolean initializeEEPROM() {}
17
18 boolean writeEEPROM() {}
19
20 void printEEPROM() {}
21
22 // ==========================================================
23 // |                          SETUP                         |
24 // ==========================================================
25
26 void setup() {}
27
28 // ==========================================================
29 // |                          LOOP                          |
30 // ==========================================================
31
32 void loop() {}
```

The `setup()` and `loop()` functions are necessary to the Arduino AVR compiler:

- `setup()` provides the program initialization: it is used to configure the libraries and to check the state of the EEPROM memory (see 3.2.3);

- `loop()` provides the main control loop: in this function the update of the displays is executed and the actual state of the analog inputs is checked (see 3.2.1).

The rest of the main program (the `main()` function) is automatically added by the Arduino library, so the final main program will be:

Listing 4: Main program from Arduino library

```
1
2 #include <Arduino.h>
3
4 int main(void)
5 {
6     init();
7
8 #if defined(USBCON)
9     USBDevice.attach();
10 #endif
11
12     setup();
13
14     for (;;) {
15         loop();
16         if (serialEventRun) serialEventRun();
17     }
18
19     return 0;
20 }
```

3.2.1 Input enable

The digital inputs are mapped to the Arduino analog inputs through the board described in section 2.3. On the board the input are kept low with a pull down resistor. This means that when the input cable is disconnected the analog inputs are not connected to anything and they are not kept low! Thus the integer value read by Arduino is a random value between 0 and 1023. This result in some weird behavior of the scoreboard like self-increasing score and timer disturbance because of the fake input.

To fix this issue I had to realize a software check that, with some limitation, control the actual values of the analog inputs and disable the input when all three analog values are different than zero. This situation is met only when the cable is disconnected or when the user pushes at least one button on each row on the control panel (each row is connected to an analog input). The latter is again a dangerous situation so the input are going to be disable in both cases. When all the three analog value are equal to zero the analog inputs are re-enabled, because this means that cable has been reconnected and the pull down resistors are working correctly. The control sequence is implemented in the `loop()` function and is shown in listing 5:

Listing 5: Main program from Arduino library

```
1
2 // Analog input not connected
3 if (!inputEnable) {
4   int homeAnalogValue = analogRead(HOME_ANALOG_INPUT);
5   // Delay before the next analogRead, for the analog-to-digital converter
6   // to settle after the last reading
7   delay(10);
8   int awayAnalogValue = analogRead(AWAY_ANALOG_INPUT);
9   delay(10);
10  int timerAnalogValue = analogRead(TIMER_ANALOG_INPUT);
11  delay(4);
12
13  // This check works only if the analog input are kept low with a pull down
14  // resistor!
15  if (homeAnalogValue == 0 && awayAnalogValue == 0 && timerAnalogValue == 0) {
16    inputEnable = true;
```

```
17    }
18  } else {
19    // Analog input connected, check for buttons pressed
20    homeButtons.checkButtons();
21    awayButtons.checkButtons();
22    timerButtons.checkButtons();
23
24    int homeAnalogValue = homeButtons.analogValue();
25    int awayAnalogValue = awayButtons.analogValue();
26    int timerAnalogValue = timerButtons.analogValue();
27
28    // The analog inputs have been disconnected: "random" value will be on
29    // each analog port (usually much greater than 0)
30    if (homeAnalogValue != 0 && awayAnalogValue != 0 && timerAnalogValue != 0) {
31      inputEnable = false;
32    }
33  }
```

At the initialization the input are disabled and after they are constantly checked by using the Arduino API and the AnalogButtons library. There is still a problems with this type of digital-to-analog inputs: when the user pushes two buttons of different row some weird behavior can happen, but after some tests of the scoreboard for many matches I can say this is not a real problem. Anyway this is not fixable with a software approach, instead a careful choice of the resistors values is necessary to avoid such problems.

3.2.2 Countdown timer

A basketball game, just like many other sports, requires a countdown timer to keep trace of the effective time in the game. This timer could be easily made using the `delay()` or `millis()` functions in the Arduino APIs, but since the precision matters I decided to use an hardware interrupt with precise calculation behind. The Arduino platform can generate some hardware interrupt using counters and the configuration is easy if you have some introduction to interrupt logic and setup in the microcontrollers field. If not you can read this good introduction [6], which end up with exactly with the configuration I used. The Atmel AVR CPU datasheet is the resource where you can find some more detailed information on the internals configurations (for the Arduino Uno the CPU is the ATmega328p [7]).

I will describe here only the configuration I used, since all the other information can be easily found at the resource I pointed to or elsewhere in the Internet. I used the Clear Timer on Compare Match, or *CTC*, interrupt on timer 1, with system clock as counter clock. The timer compares its count to a value that was previously stored in a register. When the count matches that value, the timer can either set a flag or trigger an interrupt. The interrupt will be served by an *ISR* (Interrupt Service Routine), in which the user can put the needed code. Using a prescaler of 1024 the compare match value T_c to fire the interrupt every second is 15624, considering the CPU frequency (16MHz). This value is obtained as

$$T_c = \left(\frac{T_t}{T_r}\right) - 1$$

where T_t is the timer target time, 1 s in this application, and T_r is the time resolution given the CPU clock frequency and the prescaler, obtained as

$$T_r = \frac{prescaler}{CPU_f} = 6.4 \times 10^{-5}\, s$$

Once the timer 1 reaches this value the ISR will be executed:

[6] http://www.engblaze.com/microcontroller-tutorial-avr-and-arduino-timer-interrupts/
[7] http://www.atmel.com/devices/ATMEGA328P.aspx

Listing 6: Interrupt Service Routine

```
ISR(TIMER1_COMPA_vect) {
  sec--;
  updateDisplay = true;
  saveEEprom = true;
}
```

The function manages the total seconds counter, and sets some flags to write the new time on the internal EEPROM and to refresh the displays. Thus, when the time is updated the new time is shown on the displays as soon as possible. Some care must be taken to avoid problems at runtime due to the compiler optimizations: the variables shared between the ISR and the rest of the code have to be declared `volatile`, and they have to be copied to a local variable in a interrupt free context, saving and restoring the status register (*SREG*):

Listing 7: Read shared variables in the main program

```
// Interrupt free context to update shared volatile variables
sreg = SREG;
cli();
updateDisplayLocal = updateDisplay;
saveEEpromLocal = saveEEprom;
updateDisplay = false;
saveEEprom = false;
secLocal = sec;
sei();
SREG = sreg;
```

The same special code should be used when setting and resetting the timer 1 counter value or the configuration registry, as shown in listings 8 and 9.

Listing 8: Setup of the timer interrupt

```
void setup() {
  ...
  // Setup of timer1 CTC interrupt
  // Initialize Timer1
  cli();          // disable global interrupts
  TCCR1A = 0;     // set entire TCCR1A register to 0
  TCCR1B = 0;     // same for TCCR1B

  // Set compare match register to desired timer count
  OCR1A = 15624;
  // Turn on CTC mode
  // TCCR1B |= (1 << WGM12);
  // Set CS10 and CS12 bits for 1024 prescaler
  // TCCR1B |= (1 << CS10);
  // TCCR1B |= (1 << CS12);
  // So its TCCR1B = 13 the make the timer start

  // Enable timer compare interrupt:
  TIMSK1 |= (1 << OCIE1A);
  sei();          // Enable global interrupts:
}
```

Listing 9: Use of the timer

```
case TIMER_START_STOP:
      if (timerRunning) {
        // Stop timer, save global interrupt flag and restore after
        sreg = SREG;
        cli();
        TCCR1B = 0;
        SREG = sreg;

        timerRunning = false;
        updateDisplay = true;
      } else {
        // Start/restart timer
        if (time.min == 0 && time.sec == 0) {
          return;
        }
```

```
16          sreg = SREG;
17          cli();
18          TCCR1B = 13;
19          SREG = sreg;
20
21          timerRunning = true;
22          updateDisplay = true;
23        }
24        break;
25
26      case TIMER_RESET:
27        if (!timerRunning && held) {
28          sreg = SREG;
29          cli();
30          TCNT1 = 0;
31          SREG = sreg;
32
33          time.min = TIMER_INIT_MIN;
34          time.sec = TIMER_INIT_SEC;
35          sec = TIMER_INIT_MIN * 60 + TIMER_INIT_SEC;
36        }
37        updateDisplay = true;
38        break;
```

The accuracy in time of this interrupt depends on the stability of the crystal oscillator integrated in the CPU, which is +/- 50 ppm, so it is +/- 2 minutes clock's error during one month. If the crystal is working at constant room temperature stability should be even better.

3.2.3 EEPROM wear leveling

The Arduino internal EEPROM memory has uniques electrically erasable cells, up to 1024 bytes. Each of these cell can be erases a limited number of time (erase cycle) before becoming unreliable. For the Arduino UNO r3 CPU the life is about 100000 erase cycle for *each* cell. Since the erase cycle is executed also upon a write cycle the cell write cycles are limited too. A wear leveling algorithm attempts to work around these limitations by arranging data in a such a way that re-writes are distributed evenly across the medium. In this way no single erase cell prematurely fails due to a high concentration of write cycles.

In this application the EEPROM memory is used to store the march data (score and time) and to restore them in case of power loss. I wanted to save every change in the data, so the time changes every second and this result in a write cycle for some cells in the memory. Also when the score changes it is written to the memory. Given that every basketball game is composed by at least 4 period, and each period lasts 10 minutes (effective time), the average number of writing command no the EEPROM per game is:

$$W_m \simeq 60\ seconds * 10\ minutes * 4\ periods + 100\ score\,change \simeq 2500 writings$$

I considered 100 additional writing to account for the score changes, which themselves generate an independent write cycle on the EEPROM.

Each write cycle involves the whole amount of data shown in listing 10:

Listing 10: Persistent data type

```
1 // ######################## Data types ########################
2 struct Score {
3   uint16_t home;        // 2 bytes
4   uint16_t away;        // 2 bytes
5 };
6
7 struct Time {
8   uint16_t min;         // 2 bytes
9   uint16_t sec;         // 2 bytes
10  uint16_t period;      // 2 bytes
11 };
```

```
12
13 // Pesistend data type to be written on the EEPROM
14 struct persistentData {
15   uint32_t counter;     // 4 bytes
16   Score score;          // 4 bytes
17   Time time;            // 6 bytes
18 };
```

The data type `persistentData` size is 14 bytes: so 14 cells will be written on each write command. The `counter` variable is used to keep trace of the how many writes have been executed on the cell group. If I used only one group of 14 cells the total memory life would be:

$$L = \frac{100000}{W_m} = 40\ games$$

This means that after only 40 games the first 14 cells of the EEPROM would have been written 100000 times, and thus the cells would be not reliable any more. But the EEPROM is made of 1024 cells that can be used!

In my simple algorithm I decided to use $G = \lfloor \frac{1024}{14} \rfloor = 73$ groups, in this way I can use the whole memory, starting from the first group and shifting the write cycle on to the next group when the previous one has become unreliable, after 100000 write cycles. This procedure allows the application to last much longer, in particular:

$$L = 73\ *\ \frac{100000}{W_m} = 2920\ games$$

This life is more than enough for my need so, although the algorithm can be easily improved, I used this version of the wear leveling algorithm, implemented by some functions in the main program.

Those function use the `avr/eeprom.h` library found in the Arduino include directory to read and write the EEPROM cells. are:

`initializeEEPROM()` :

Used to initialize the algorithm to write the EEPROM starting from the first not out of order cell, i.e. its life is not over. This function scans the whole memory and find the first usable cell, than the rest of the algorithm will write starting from this location. If the memory is full, i.e. there are no remaining cells, the application will not save anything and the related functions will be not used. This functions read also the last saved values if present, which can be restored at any time using the control panel in setup mode. The values are stored in RAM so that the application can still save the data when needed without overwrite the restore data in the EEPROM.

Listing 11: initializeEEPROM() function

```
1  boolean initializeEEPROM() {
2    persistentData data;
3
4    uint16_t size = sizeof(data);
5
6    uint16_t readCounter = (uint16_t) EEPROM_SIZE / size;
7
8    for (uint16_t i = 0; i < readCounter; i++) {
9      eeprom_read_block((void*) &data, (void*) (size * i), sizeof(data));
10
11     if (data.counter < EEPROM_MAX_WRITE) {
12       if (data.counter == 0 && i > 0) {
13         // Read from previous cell, just reached end of life
14         i--;
```

```
15          eeprom_read_block((void*) &data, (void*) (size * i), sizeof(data));
16        }
17        counterEE = data.counter;
18        offsetEE = size * i;
19        dataEE = data;
20        return true;
21      }
22    }
23
24    // EEPROM full (end of life), read last saved data
25    dataEE = data;
26    return false;
27  }
```

`writeEEPROM()` :

Used to write the actual data to the EEPROM when requested by the application. The function writes the data and increment the actual write counter in the cell group. If the is no cells available the function simply returns.

Listing 12: writeEEPROM() function

```
1  boolean writeEEPROM() {
2    persistentData data;
3    uint16_t size = sizeof(data);
4
5    counterEE++;
6
7    if (counterEE > EEPROM_MAX_WRITE) {
8      offsetEE += size;
9      counterEE = 1;
10   }
11
12   if (offsetEE >= EEPROM_SIZE) {
13     return false;
14   }
15
16   data.counter = counterEE;
17   data.score = score;
18   data.time = time;
19
20   eeprom_write_block((void*) &data, (void*) offsetEE, sizeof(data));
21
22   return true;
23 }
```

The are two other service functions: `readEEPROM()` and `printEEPROM()`: the former reads the data in the EEPROM at a specified offset, and the latter prints the EEPROM content to the serial interface, to show the actual state of the memory, for debug purpose.

3.3 Volleyball mode

In a recent release (v1.0.2) the volleyball mode has been implemented. This was done on requested from a volleyball coach, so I tried to adapt the software without touching the hardware (the scoreboard is already mounted on the wall by now). The new mode simply changes the meaning of some displays in a way that is better suited for a volleyball match.

Figure 11 shows the new configuration: the two upper display groups show the score for each team. The central one shows the current set in the match, and the two at the bottom show the sets won by each team. In listing 13 is shown the code responsable for the configuration switch: using the DisplayGroup library API the main program cleans the displays configuration when the volleyball mode is activated (row 5) and inserts the new configuration (rows 7-16). The same procedure is carried out when the same button is pressed again, switching back the configuration to basketball mode.

Figure 11: ScoreBoard in volleyball mode

Listing 13: Volleyball mode activation and deactivation

```
1  volleyMode = !volleyMode;
2
3  if (volleyMode) {
4    // Volleyball mode
5    disManager.clearGroups();
6
7    disManager.addGroup(0, 2, &vScore.home, gDigits7, sizeof(gDigits7));
8    disManager.addGroup(1, 2, &vScore.away, gDigits7, sizeof(gDigits7));
9    disManager.addGroup(2, 1, &vSets.actSet, gDigits7, sizeof(gDigits7));
10   disManager.addGroup(3, 1, NULL);
11   disManager.addGroup(4, 1, &vSets.homeSet, gDigits4, sizeof(gDigits4));
12   disManager.addGroup(5, 1, &vSets.awaySet, gDigits4, sizeof(gDigits4));
13   disManager.addGroup(6, 1, NULL);
14
15   disManager.enableGroup(3, false);
16   disManager.enableGroup(6, false);
17
18 } else {
19   // Basketball mode
20   disManager.clearGroups();
21
22   disManager.addGroup(0, 2, &bScore.home, gDigits7, sizeof(gDigits7));
23   disManager.addGroup(1, 2, &bScore.away, gDigits7, sizeof(gDigits7));
24   disManager.addGroup(2, 1, &time.period, gDigits7, sizeof(gDigits7));
25   disManager.addGroup(3, 2, &time.min, gDigits4, sizeof(gDigits4));
26   disManager.addGroup(4, 2, &time.sec, gDigits4, sizeof(gDigits4));
27 }
```

The input mode must be setupMode and one button will allow the user to switch between basketball mode and volleyball mode. Two groups are disabled (rows 15-16) because they are not used to show the number of sets won by the two teams (a digit is enough for each team!). The switch can be carried out on the fly, at runtime, and all the information regarding on mode or the other are kept back in the respective variables; thus when the user switches back to one mode he will find the displays in the same state they were when he left that mode. This works only if there hasn't been any power cycle in between, because the backup on the EEPROM memory works only for basketball mode.