SCOREBOARD PROJECT

Electronic board
with 7-segments displays
and Arduino

version 0.8

Revamped with wireless controller!!

Gionata Boccalini

November 9, 2014

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 they 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.

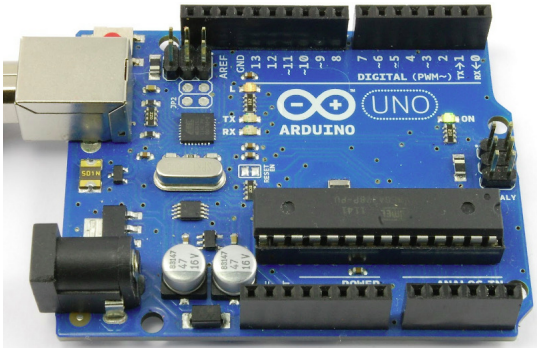


Figure 2: Arduino UNO platform

- 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/

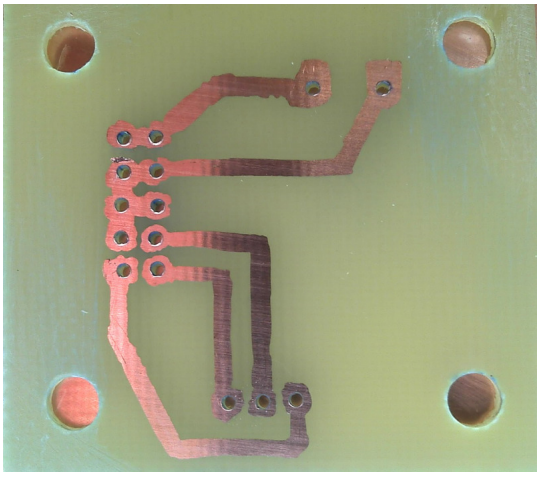


Figure 3: Homemade connection board without components

- 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:

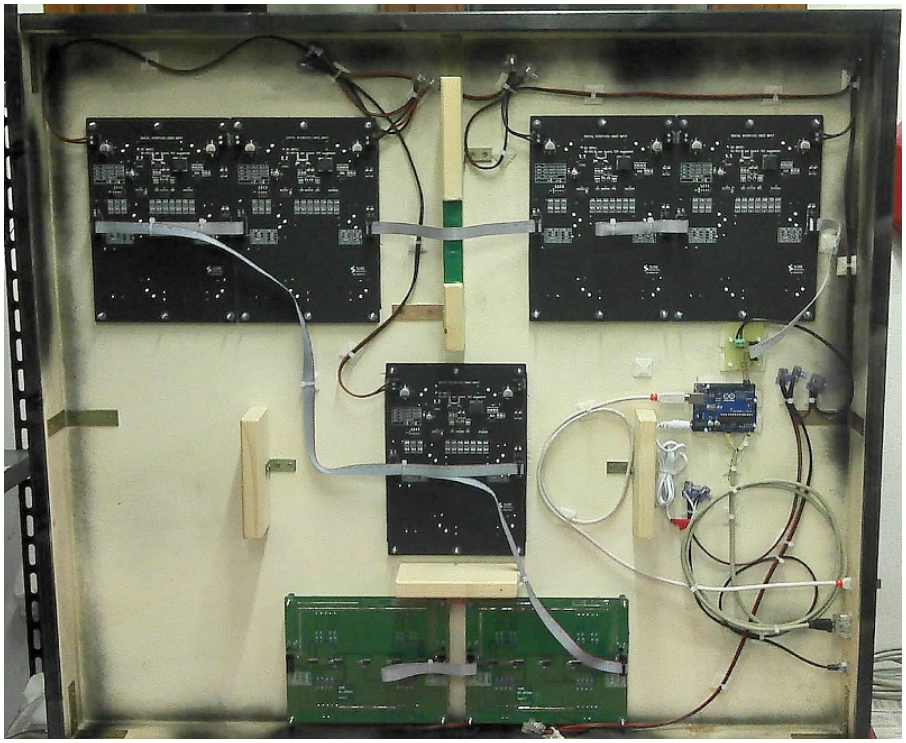


Figure 4: Back side of the board, with Arduino and electrical connections

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.

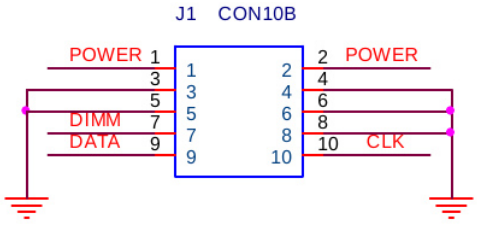


Figure 5: Power and data connection on the 7-segments board

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.

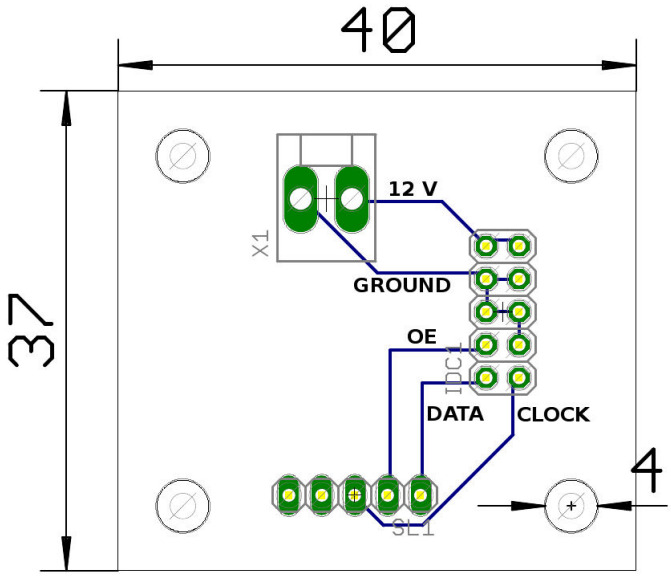


Figure 6: Connection board schematic

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.

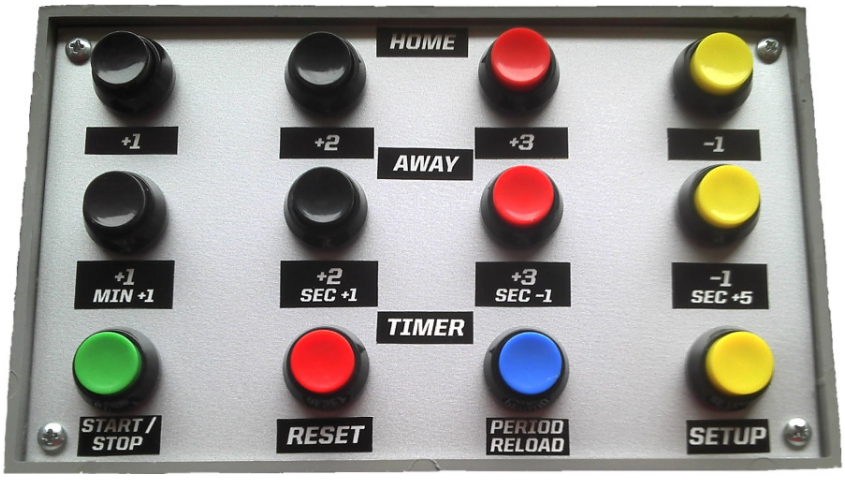


Figure 8: Control panel front view

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:

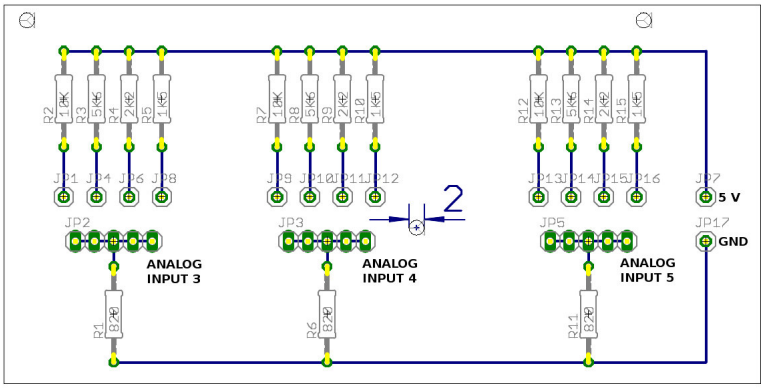


Figure 9: Control panel connection board

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:

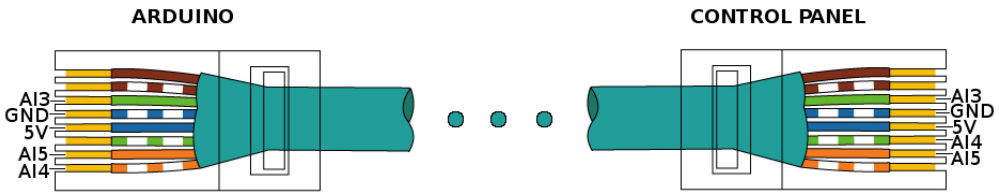


Figure 10: Control panel connection board

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.

2.4 Wireless Extension

The wireless communication was not needed by the time I realized the first version of the scoreboard, but then, after 2 years of good use of the system, my basketball team was able to switch championship.. to an upper level! So new championship means new rules! The space between the field border and the wall on the scoreboard side was not enough for the new, more

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

strict, rules. We needed to move the service desk to the other side of the field. The cable between the control panel and the scoreboard would have been way too long and difficult to place, so this is when the wireless saga began!

The research for the right components took me months, as I was looking for all these features;

- fast enough and reliable communication, even with many disturbances (cellphones, etc..);
- wireless range cover (20-30 meters);
- small components to fit in the control panel I had already, without changing anything inside;
- long battery life;
- easy realization, homemade as always!

I found the XBee RF Modules [5] to meet the unique needs of low-cost, low-power wireless sensors, since they require minimal power and provide reliable delivery of data between devices. For the purpose of this application I used the Xbee 802.15.4 Series 1 module, which are more than enough for me, and not too much expensive:

- Indoor/Urban range up to 100 feet (30 m)
- TX Peak Current: 45 mA (@3.3 V)
- RX Current: 50 mA (@3.3 V)
- Transmit Power: 1 mW (0 dBm)
- Receiver Sensitivity: -92 dBm

These Xbee module are also equipped with an on board analog to digital converter, so that I could connect my buttons board directly to the Xbee: all the hard work of collecting the samples and sending them over the air is done by the module! The Xbee are also really easy to configure, and they have a lot of options.

I decide to use a power switching IC to optimize the battery power usage, and I wanted to use two standard AA sized Alkaline battery. The power consumption is really low during data transmission but I wanted to keep everything even lower and use some of the intelligent Xbee features like hibernation (*sleep mode*). These two requirements gave me the possibility to learn a lot on power management in electronic devices, and even on using and programming small MCU not related to Arduino.

2.4.1 Transmitter board

The picture 11 shows the schematic circuit for the transmitter board I designed. It is base on some main component and a some auxiliary components that are needed to make everything work smoothly.

The real board is manufactured by the Seeed Studio Fusion PCB service [6], (yes... this one is too cluttered to be made by hand), really fast and easy to use service from China. The board is in picture 12, the quality is good and the price is well worth the product you'll get. The

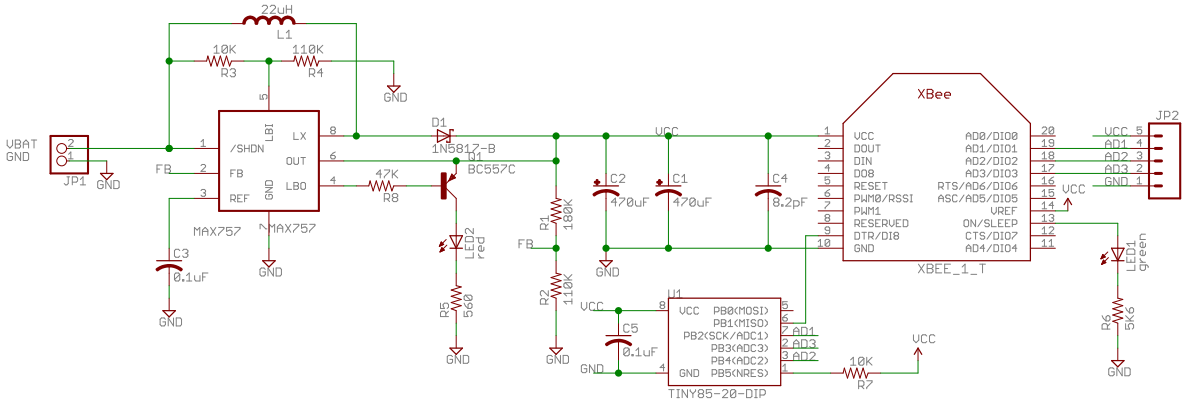
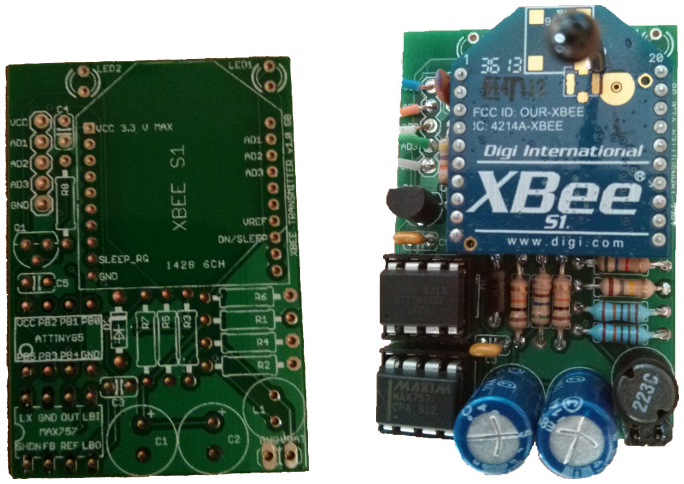


Figure 11: Transmitter board schematic



(a) **(b)**

Figure 12: Transmitter board from Seeed Studio (a), and board with components (b)

schematic and board file made with Eagle are available in the Google Code site of this project, in the download page.

The power converter is a Maxim MAX757 switching power regulator[7], connected as shown in the datasheet, with a LED connected on the LBO output, to signal the end of battery life with some time in advance. The threshold for the battery voltage to signal the battery as empty is set with R3 and R4, all the details are given in the datasheet.

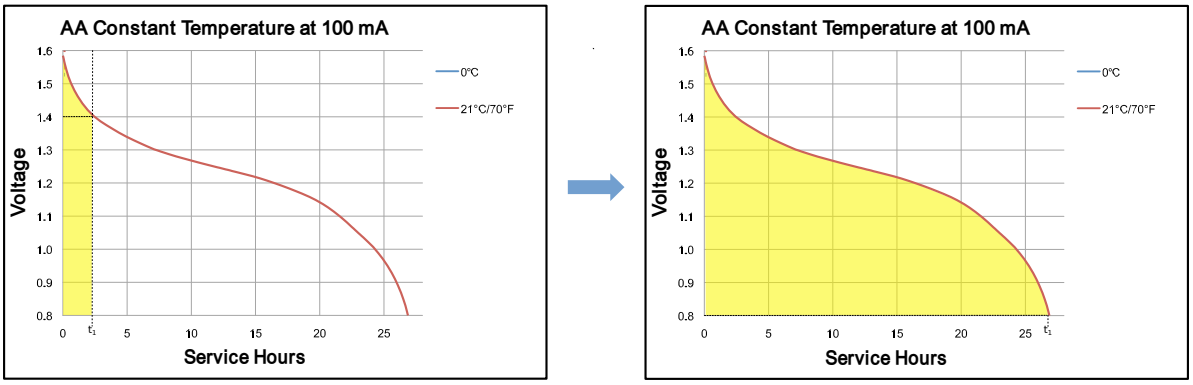
Figure 13 shows the circuit working region with and without the step-up regulator: some digital component like Xbee can work with not less than 2.8V as Vcc, this makes 1.4V per battery (with two batteries connected in series). In figure 13:A the minimum level of 1.4V is reached at t_1, which is about 2.5 hours of running time, at constant 100mA discharge current, 21 C. With a power regulator like the MAX757, that can work down to 0.7V of input supply voltage, the working region is extended like in 13:B. The level 0.8V is reached at t_1, almost at 28 running hours, and the regulator is still working in good conditions, with a constant output voltage of 3.3V to supply the Xbee and the rest of the circuit for much more time. Figure 13 is just for

[5] http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/point-multipoint-rfmodules/xbee-series1-module

[6] https://www.seeedstudio.com/service/index.php?r=pcb

[7] http://www.maximintegrated.com/en/products/power/switching-regulators/MAX757.html

demonstration purpose.



(a) (b)

Figure 13: Typical alkaline battery discharge curve, with working region for the circuit: (a) without power regulator, (b) with power regulator

The typical auxiliary circuit for the regulator is better explained in the datasheet, for the purpose of this application I have just added a transistor controlled by the LBO output, to switch on the LED without current drain from the IC itself. The LBO output is only capable of 2mA, not enough to turn on the LED2 to be visible.

The micro-controller in figure 12 is a Atmel AVR ATTiny 85V[8], ultra low voltage MCU. It is used to wake up the Xbee module only when necessary, so that the Xbee can sleep most of the time. When the Xbee is Pin Hibernate mode uses only 10uA, much less than transmitting current 45mA. In this mode the Xbee can be waken up only by setting low the line on input 9 $\overline{\text{DRT}}$/SLEEP_RQ, and this is the job of the micro-controller. The MCU watch the 3 analog inputs to detect a pressed button, and wakes up the Xbee. The Xbee will sample the data from the same analog channels and send it over the RF link. The code running on the MCU is described in listing 14.

The LED1 is used to signal that the Xbee is awake.

The JP2 connector is directly connected to the board in figure 9, providing power (3.3V) and retrieving the three analog values from the buttons. These lines will be sampled by the Xbee and transmitted to the other Xbee in the scoreboard in a *autonomous way*, once the Xbee are configured correctly.

The capacitors C1, C2, C4 and C5 are filters on the power supply; they are needed to provide a constant, smooth voltage to the integrated components. The regulator itself works in a region were it's switching frequency is about 100kHz (datasheet) but the Xbee requires at least 500kHz as a good recommendation on the power supply. I couldn't find another regulator with the same features and higher switching frequency, so I added some big electrolytic capacitors to minimize the voltage ripple. The analog values are almost constant and all the ICs are working correctly!

2.4.2 Receiver board

The picture 14 shows the schematic circuit for the receiver board, which is mounted on the back of the scoreboard, near Arduino and the displays. The board is home made with a really nice technique used to create PCBs, called Press-n-Peel[9], which is easy and you can get beautiful PCB in short time.

[8] http://www.atmel.com/devices/attiny85.aspx
[9] http://www.techniks.com/how_to.htm

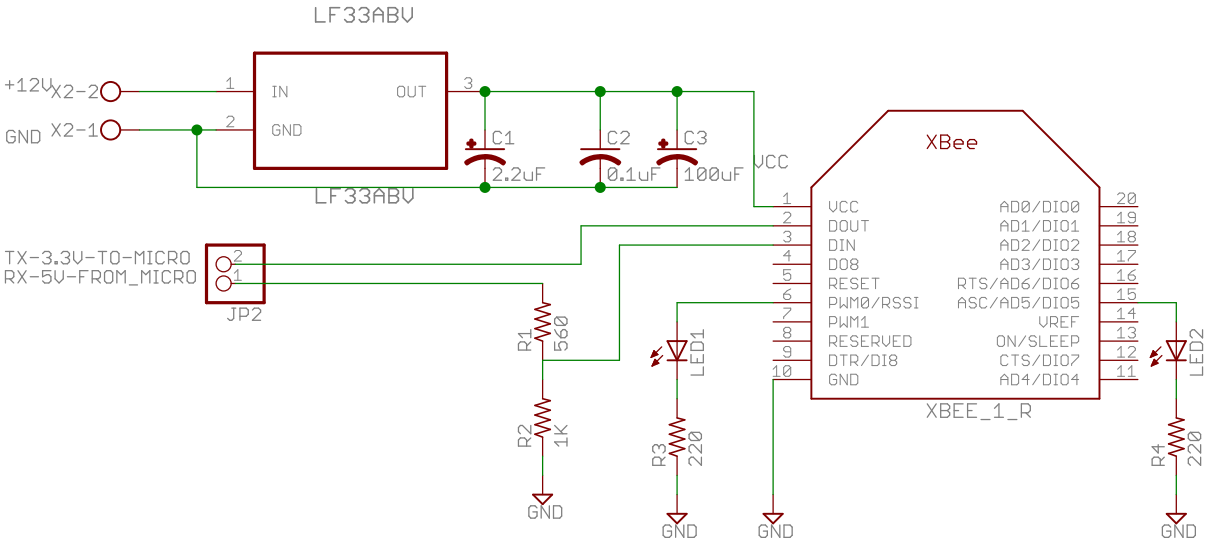
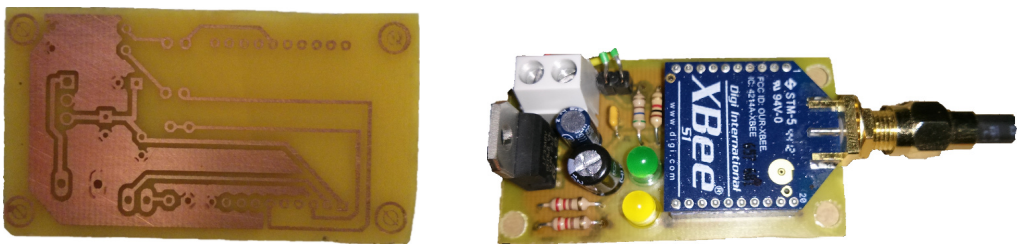


Figure 14: Receiver board schematic

The board is made of a low dropout linear voltage regulator, this time with input up to 16V, to use the power supply voltage on the scoreboard which is 12V. The output is fixed 3.3V to supply the Xbee module, with filtering capacitor as shown in the datasheet. The Xbee DOUT line is directly connected to the serial input pin of the Arduino, while the DIN line from Arduino [0, 5V] is buffered with a voltage divider, to take the signal in the range [0, 3.3V] to be compatible with the Xbee. LED1 is used to show the strength of the received signal (RSSI), and LED2 is used to indicate that this Xbee is associated with another node in the network.

The complete board is shown in picture 15, the Xbee used here has the RPSMA connector, with a long WIFI antenna, which is long enough to take the receiving part outside of the scoreboard metal cage. This Xbee module with this type of antenna is a little bit more expensive than the standard ones, but it was the most easy way I found to avoid problems like disturbances or signal degradation at the receiving site.



(a) **(b)**

Figure 15: Receiver board, bottom layer (a), and board with components (b)

2.4.3 Xbee configuration - transmitter

The transmitter Xbee is able to sample the ADC data from the input lines, build the serial data frame according to the frame type RX Packet, 16-bit Address IO (API Identifier Value: 0x83), and send the frame to the UART. If the $\overline{\text{DRT}}$/SLEEP_RQ line is high the Xbee will switch to the configured sleep mode once has finished the packet transmission. To achieve this kind of behavior the configuration is set as in table 1, where only non-default parameters are shown.

The sleep mode is Pin Hibernate, which provides the best energy save, and the time before sleep

Transmitter Xbee configuration			
Section	**Parameter**	**Value**	**Description**
Network & Security	CH	10	Channel
	ID	3340	PAN ID
	DH	13A200	Destination Xbee address high part
	DL	40A191C7	Destination Xbee address low part
	MY	2	16Bit source address
Sleep modes	SM	1	Sleep mode
	ST	1	Time before sleep
	DP	14	Disassociated cyclic sleep period
Serial interfacing	BD	3	Interface data rate
I/O Settings	D5	0	DIO5 configuration
	D3	2	DIO3 configuration
	D2	2	DIO2 configuration
	D1	2	DIO1 configuration
	IT	2	Sample before transmission
	IR	28	Sample rate

Table 1: Transmitter Xbee configuration parameters

is as short as possible. DIO1-2-3 are used as analog input, sample are taken every 64ms (28 Hex) when Xbee is awake, and a package is sent every 2 samples collected, so every 128ms. This is fast enough to provide sufficient response time when a button is pressed: the time needed for two consecutive buttons to get to Arduino is usually less the 300ms, enough to stop and start the timer with the precision I wanted for this application.

2.4.4 Xbee configuration - receiver

The receiver Xbee is configured as the network coordinator (even if this is not a meshed, multinode network...), it simply listens on the radio interface for packages from the transmitting node. When a frame is received the Xbee sends it to the Arduino via a simple serial interface. The configuration is shown in table 2.

Receiver Xbee configuration			
Section	**Parameter**	**Value**	**Description**
Network & Security	CH	10	Channel
	ID	3340	PAN ID
	DH	0	Destination Xbee address high part
	DL	0	Destination Xbee address low part
	MY	1	16Bit source address
	CE	1	Coordinator enable
Sleep modes	DP	14	Diassociated cyclic sleep period
Serial interfacing	BD	3	Interface data rate

Table 2: Receiver Xbee configuration parameters

The serial interface data rate to Arduino is limited to 19200 baud, because of some problem with clock synchronization between the Xbee and Arduino. This speed was enough for my needs, so

I decided to keep the slower (and more reliable) speed.

Some more information regarding this timing problems can be found at the Digi forum[10], and in other unofficial resources on the Internet.

2.4.5 ATtiny configuration

The ATtiny85 MCU on the transmitter board is configured to minimize the power consumption, and is used to monitor the analog signal more often than the Xbee, with much less current drained. The configuration is done via software, as the following code shows:

Listing 1: ATtiny85V configuration

```
1  void setup() {
2    pinMode(WAKE_OUTPUT, OUTPUT);
3    digitalWrite(WAKE_OUTPUT, HIGH);
4
5    // Setup watchdog timeout
6    setup_watchdog(0);
7
8    // Set the internal registers to reduce power consumes
9    PRR &= ~(1<<PRTIM1);          // Shut down the timer1
10   ACSR = (1<<ACD);              // Shut down the analog comparator
11   MCUCR |= (1<<BODS);           // BOD disabled
12
13 } // End of setup
```

Timer1, analog comparator and BOD protection are tunerd off by defautl. When the MCU wakes up, the ADC convert is turned on, and then back off before the MCU sleeps again.

The sleep mode is `SLEEP_MODE_PWR_DOWN`: in this mode the oscillator is stopped, while the external interrupts, the USI start condition detection and the watchdog continue operating. The watchdog interrupt is used to wake the MCU every a predefined amount of time, in this case 16ms, through the function `setup_watchdog(0)`, which sets the watchdog timeout in the MCU registers.

[10] http://www.digi.com/support/forum/4787/using-the-xbee-at-115-200-baud-updated-16-march-2010

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 Dependencies

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 2: 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 AnalogButtonsGeneral, which is a slighlty modified version of AnalogButtons[11], and is used to control many digital inputs through few analog inputs from the Xbee analog to digital interface.

This library has been modified to export the analog value to be read outside of the it's own code, into the application source. This is necessary to use, for example, a wireless sensor that reads the analog values and sends them on a serial connection. In general this is a required step every time the sensors or buttons are not physically connected to the Arduino analog input.

The configuration for this library is identical to the one of the original AnalogButtons library:

Listing 3: AnalogButtonsGeneral configuration

```
1 // Patch cable
2 const uint16_t bAVal[10] = { 50, 95, 110, 170, 200, 295, 310, 390 };
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]);
```

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

```
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 void setup() {
24   ...
25
26   // Input Buttons
27   homeButtons.addButton(b2);
28   homeButtons.addButton(b1);
29   homeButtons.addButton(b3);
30   homeButtons.addButton(b4);
31
32   awayButtons.addButton(b5);
33   awayButtons.addButton(b6);
34   awayButtons.addButton(b7);
35   awayButtons.addButton(b8);
36
37   timerButtons.addButton(b9);
38   timerButtons.addButton(b10);
39   timerButtons.addButton(b11);
40   timerButtons.addButton(b12);
41 }
```

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 the control logic for each buttons is implemented. Some of the buttons have an `HELD_DURATION` options: these inputs must be held for some seconds to become active.

The third dependence library has been introduced along with the support for wireless communication, and is the SoftwareSerial[12] library, which is base on the NewSoftwareSerial library [13]. The communication between the receiving Xbee and Arduino is done via a serial link, supported by Arduino with the hardware serial port (digital IO 0 - 1), but those IO pins are needed also to communicate with a PC for example to debug the program. I didn't want to loose the *only* debug chance that I had (the serial interface) so I used the software serial library which is able to emulate the RX and TX pins on any Arduino pin. The configuration is really easy:

Listing 4: SoftwareSerial configuration

```
1 // Xbee serial communication
2 #define SOFT_SERIAL_IN          6
3 #define SOFT_SERIAL_OUT         7
4
5 // Software serial input port (from Xbee)
6 SoftwareSerial xbeeSerial(SOFT_SERIAL_IN, SOFT_SERIAL_OUT);
7
8 void setup() {
9   ...
10
11  // Set the data rate for the SoftwareSerial port
12  xbeeSerial.begin(19200);
13 }
```

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 5 there are three callbacks to handle the analog input, one to handle the timer interrupt, and some functions to manage the EEPROM memory.

[12] http://arduino.cc/en/Reference/SoftwareSerial
[13] http://arduiniana.org/libraries/NewSoftSerial/

Listing 5: Main program structure

```
1
2  void configureAnalogB();
3
4  // Handle home buttons pressed
5  void handleHomeButtons(int id, boolean held);
6
7  // Handle away buttons pressed and setup mode
8  void handleAwayButtons(int id, boolean held);
9
10 // Handle timer buttons pressed and setup mode
11 void handleTimerButtons(int id, boolean held);
12
13 // ##########################################################
14 // ####### EEPROM wear leveling algorithm functions ########
15 // ##########################################################
16
17 // EEPROM wear leveling algorithm prototypes
18 // Seek for the first writable cells in the EEPROM  i.e. the first cell counter
19 // with value smaller than EEPROM_MAX_WRITE
20 boolean initializeEEPROM();
21
22 // Write to the EEPROM cell with offset "offsetEE" and counter "counterEE" the
23 // actual value of score and time variables.
24 boolean writeEEPROM();
25
26 // Read from the EEPROM cell with offset "offsetEE" and counter "counterEE" the
27 // stored value of score and time variables.
28 void readEEPROM();
29
30 // Prints on the serial interface the contents of all the EEPROM cells
31 void printEEPROM();
32
33 // Buzzer managament: end of period or manual activation
34 void buzzer(boolean activation);
35
36 void setupDiplay();
37
38 // Handle serial communication with Xbee
39 void xbeeSerialCom(boolean debug);
40
41 // ========================================================
42 // |                          SETUP                       |
43 // ========================================================
44
45 void setup() {}
46
47 // ========================================================
48 // |                          LOOP                        |
49 // ========================================================
50
51 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 6: 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

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

The new version of the application itself uses an Xbee to read the actual analog values on the buttons lines, and the Xbee is constantly connected to the control panel board, so this is not a problem any more!

Anyway, the Arduino source code must constantly read analog values from the sensor for the input to work correctly, but this means that the Xbee radio should be always active. The battery drain in this case is unacceptable. To preserve battery life, the Xbee is only waken up when necessary, and is kept awake for the minimum amount of time needed to transmit also the logical value 0 when the user releases all the buttons. In this way the code running on Arduino can reset it internal condition and be ready for the next button that will be pressed.

There is still a problems with this type of digital-to-analog inputs: when the user pushes two or more buttons of different row at the same time some weird behavior can happen, but after some tests of the scoreboard for many games 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 [14], 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 [15]).

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

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

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:

Listing 7: 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 8: 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 9 and 10.

Listing 9: 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);
```

```
15   // TCCR1B |= (1 << CS12);
16   // So its TCCR1B = 13 the make the timer start
17
18   // Enable timer compare interrupt:
19   TIMSK1 |= (1 << OCIE1A);
20   sei();          // Enable global interrupts:
21 }
```

Listing 10: Use of the timer

```
1 case TIMER_START_STOP:
2       if (timerRunning) {
3         // Stop timer, save global interrupt flag and restore after
4         sreg = SREG;
5         cli();
6         TCCR1B = 0;
7         SREG = sreg;
8
9         timerRunning = false;
10        updateDisplay = true;
11      } else {
12        // Start/restart timer
13        if (time.min == 0 && time.sec == 0) {
14          return;
15        }
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 11:

Listing 11: Persistent data type

```
// ######################## Data types ########################
struct Score {
  uint16_t home;        // 2 bytes
  uint16_t away;        // 2 bytes
};

struct Time {
  uint16_t min;         // 2 bytes
  uint16_t sec;         // 2 bytes
  uint16_t period;      // 2 bytes
};

// Pesistend data type to be written on the EEPROM
struct persistentData {
  uint32_t counter;     // 4 bytes
  Score score;          // 4 bytes
  Time time;            // 6 bytes
};
```

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 12: initializeEEPROM() function

```
boolean initializeEEPROM() {
  persistentData data;

  uint16_t size = sizeof(data);

  uint16_t readCounter = (uint16_t) EEPROM_SIZE / size;

  for (uint16_t i = 0; i < readCounter; i++) {
    eeprom_read_block((void*) &data, (void*) (size * i), sizeof(data));

    if (data.counter < EEPROM_MAX_WRITE) {
      if (data.counter == 0 && i > 0) {
        // Read from previous cell, just reached end of life
        i--;
        eeprom_read_block((void*) &data, (void*) (size * i), sizeof(data));
      }
      counterEE = data.counter;
      offsetEE = size * i;
      dataEE = data;
      return true;
    }
  }

  // EEPROM full (end of life), read last saved data
  dataEE = data;
  return false;
}
```

`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 13: writeEEPROM() function

```
boolean writeEEPROM() {
  persistentData data;
  uint16_t size = sizeof(data);

  counterEE++;

  if (counterEE > EEPROM_MAX_WRITE) {
    offsetEE += size;
    counterEE = 1;
  }

  if (offsetEE >= EEPROM_SIZE) {
    return false;
  }

  data.counter = counterEE;
  data.score = score;
  data.time = time;

  eeprom_write_block((void*) &data, (void*) offsetEE, sizeof(data));

  return true;
}
```

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 Wireless Extension

This wireless version of the project requires some custom functions in both the ATtiny MCU on the transmitter and on the Arduino at the receiving site.

3.3.1 ATtiny code

The ATtiny main code is shown in listing 14: it simply setup a pin in output mode to put the Xbee to sleep, setup the watchdog interrupt, and setup some internal registers value to reduce power consumption. The `loop` function scans for some button pressed with a minimum analog threshold, and in case at least one button is pressed, the Xbee is waken up with a low output on the `WAKE_OUTPUT` pin.

Listing 14: ATtiny main code

```
1  void system_sleep() {
2    cbi(ADCSRA, ADEN);                    // Switch Analog to Digitalconverter OFF
3    set_sleep_mode(SLEEP_MODE_PWR_DOWN);  // Sleep mode is set here
4    sleep_enable();
5    sleep_mode();                         // System actually sleeps here
6    sleep_disable();                      // System continues execution here when
7                                          // watchdog timed out
8    sbi(ADCSRA, ADEN);                    // Switch Analog to Digitalconverter ON
9  }
10
11 // Watchdog Interrupt Service Routine
12 ISR(WDT_vect) {
13   wdCount++;
14 }
15
16 // =======================================================
17 // |                        SETUP                        |
18 // =======================================================
19
20 void setup() {
21   pinMode(WAKE_OUTPUT, OUTPUT);
22   digitalWrite(WAKE_OUTPUT, HIGH);
23
24   // Setup watchdog timeout
25   setup_watchdog(0);
26
27   // Set the internal registers to reduce power consumes
28   PRR &= ~(1<<PRTIM1);                  // Shut down the timer1
29   ACSR = (1<<ACD);                      // Shut down the analog comparator
30   MCUCR |= (1<<BODS);                   // BOD disabled
31
32 } // End of setup
33
34 // =======================================================
35 // |                        LOOP                         |
36 // =======================================================
37
38 void loop() {
39
40   system_sleep();                       // Send the unit to sleep
41
42   boolean pressedHome = (analogRead(HOME_ANALOG_INPUT) > MIN_ANA_TH);
43   boolean pressedAway = (analogRead(AWAY_ANALOG_INPUT) > MIN_ANA_TH);
44   boolean pressedTimer = (analogRead(TIMER_ANALOG_INPUT) > MIN_ANA_TH);
45
46   if (pressedHome || pressedAway || pressedTimer) {
47     digitalWrite(WAKE_OUTPUT, LOW);
48     wdCount = 0;
49   } else if (wdCount >= WD_CMP_MATCH) {
50     digitalWrite(WAKE_OUTPUT, HIGH);
```

```
51      wdCount = 0;
52    }
53
54  } // End of loop
```

The `wdCount` variable is incremented by the watchdog interrupt; in case no buttons is pressed, after `WD_CMP_MATCH` interrupt fired the MCU will write high on the output, letting the Xbee sleep again. The watchdog timeout and the `WD_CMP_MATCH` count are chosen in such a way that the Xbee will stay awake for the necessary amount of time to sample the analog input, build the package and send it. The expression is:

$$wdCount * wdInterval \geq xBeeTX * xBeeIR$$

Where $xBeeTX$ and $xBeeIR$ are the xBee configuration parameters TX (sample before transmission) and IR (sample rate in ms). In this application the values are: $5 * 16ms \geq 2 * 40ms$. Thus, after the buttons have been released, the Xbee will be awake for at least other 80ms, and will send another package with the sampled values equal to 0. In this way the Arduino code can recognise that no button is pressed any more.

3.3.2 Arduino code

The Arduino main code has been modified to add a function, `xbeeSerialCom()`, used to parse serial data coming from the Xbee. The listing 15 shows the method without debugging flags:

Listing 15: Arduino main code: xbeeSerialCom function

```
1  void xbeeSerialCom() {
2    int xbeeDel = 0x7E;
3    int car = 0;
4    int frameType = 0;
5    int sourceAdr = 0;
6    int sampleN = 0;
7    int analogReadings[N_ANALOG_INPUT];
8
9    if (xbeeSerial.available() >= 18) {
10
11     // Seek for the frame delimiter "xbeeDel", -1 indicates empty serial buffer
12     while ((car = xbeeSerial.read()) != xbeeDel) {
13       if (car == -1) {
14         return;
15       }
16     }
17
18     // Reading API frame type 83 header
19     for (byte i = 2; i <= 8; ++i) {
20       car = xbeeSerial.read();
21
22       if (i== 4) {
23         frameType = car;
24         if (frameType == 97) {
25           return;
26         }
27       } else if (i == 6) {
28         sourceAdr = car;
29       }
30     }
31
32     // Reading header of IO data frame
33     // Reading number of samples
34     sampleN = xbeeSerial.read();
35
36     // Reading IO channel enabled mask: not used
37     car = xbeeSerial.read();
38
39     // Reading 0 byte
40     car = xbeeSerial.read();
41
```

```
42      // Reading paylod: only ADC values
43      int msb[N_ANALOG_INPUT], lsb[N_ANALOG_INPUT];
44      for (byte j = 0; j < sampleN; ++j) {
45
46        for (byte i = 0; i < N_ANALOG_INPUT; ++i) {
47          msb[i] = xbeeSerial.read();
48          lsb[i] = xbeeSerial.read();
49          analogReadings[i] = lsb[i] + (msb[i] * 256);
50        }
51
52        // Analog input received, check for buttons pressed
53        // only if the Xbee source address is recognized
54        if (sourceAdr == XBEE_SOURCE_ADR) {
55          homeButtons.checkValue(analogReadings[0]);
56          awayButtons.checkValue(analogReadings[1]);
57          timerButtons.checkValue(analogReadings[2]);
58        }
59      }
60      // Reading checksum - at the end of the frame
61      car = xbeeSerial.read();
62    }
63  }
```

Rows 12-16 read every character until the start frame delimiter, `0x7E`. Rows 19-30 read every character in the API frame header. The IO data parsing begins at line 32: number of samples, digital IO mask, and analog values in the form of 2 byte for each analog input. On line 54 a security check is made, to verify the source Xbee 16bit address. On lines 55-57 the `AnalogButtonsGeneral::checkvalue()` method is called, once for each analog line, to check the current analog value and if a buttons is pressed.

3.4 Volleyball mode

In a recent release sthe volleyball mode has been implemented. This was done upon request from the 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 16: ScoreBoard in volleyball mode

Figure 16 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 16 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.

Listing 16: 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.