Vrije Universiteit Brussel

OSSEC mini project

Real-Time Operating System on Arduino

Author:
Nicolas Pantano

Professor:
Pr. Martin Timmerman

Year 2011 - 2012

Contents

1 Introduction **2**

2 Real-Time Operating System (RTOS) **2**
2.1 What is a RTOS? . . . 2
2.2 Why do we need a RTOS? . . . 3
2.2.1 Example . . . 3
2.3 Multitasking and scheduler . . . 3
2.3.1 Cooperative multitasking . . . 4
2.3.2 Preemptive multitasking . . . 4

3 Demonstration application **5**
3.1 Introduction . . . 5
3.2 Arduino . . . 6
3.3 FreeRTOS . . . 7
3.4 Implementation . . . 8
3.4.1 Hardware . . . 8
3.4.2 Software . . . 9
3.5 Results . . . 14

4 Conclusion **15**

A Source code **18**

1 Introduction

Most of control systems use now real-time operating systems (RTOS) to ensure temporal constraint and reliability. RTOS are used in most common systems like internal calculators of vehicles, nuclear power plants, telecommunication systems and so forth,... RTOS are used everywhere. A RTOS is needed when we would that a task is executed within a precise time interval, or when you would that your system reacts in the time constraint you imposed to an external system.

This report, will show utility of a real-time OS with a practical application. The demonstration application is consists of an Arduino connected to three captors and relied by USB to a personal computer, which can interact with the Arduino.

2 Real-Time Operating System (RTOS)

In this section we will roughly explain the mechanism of a real-time operating system, why do we need a RTOS and in which case.

2.1 What is a RTOS?

Before seeing what a real-time OS is, let's explain the difference between a general purpose operating system and a real-time operating system. Windows, Mac OS and Linux are general purpose OS, they are designed to run multiple applications and to give the illusion to the user that they are all running simultaneously. They are named as non-deterministic OS, in other words, the temporal constraint is not the most important factor.

In real-time OS, it's a bit different. Real-time OS are specially designed to run applications with very precise timing and a high degree of reliability [1]. Thus, RTOS are used when the temporal constraint in a system is very important.

There are different kinds of RTOS, *soft*, *firm* and *hard*. In order to characterize them, we will introduce the *jitter*. The Jitter is a measure of how much the execution time of a task differs over subsequent iterations. Real-time operating systems are optimized to minimize the jitter [1].

- *Soft real-time OS* are characterized with a high jitter, soft real-time OS are used when the time constraint in which tasks must be executed is less important. Typically used in systems when the risk that a task is not executed in time is not critical.

- *Hard real-time OS* have a less jitter than soft real-time OS, in this kind of RTOS the time constraint is very important and if a task is not executed in time, the result is no more useful and the system doesn't work anymore. Hard real-time OS are used, for example, in ABS (anti-lock braking system) or Airbags.

- *Firm real-time OS* are located between soft and hard real-time OS. The Jitter is less than for soft RTOS and higher than for hard RTOS.

  [1, 2, 3, 4]

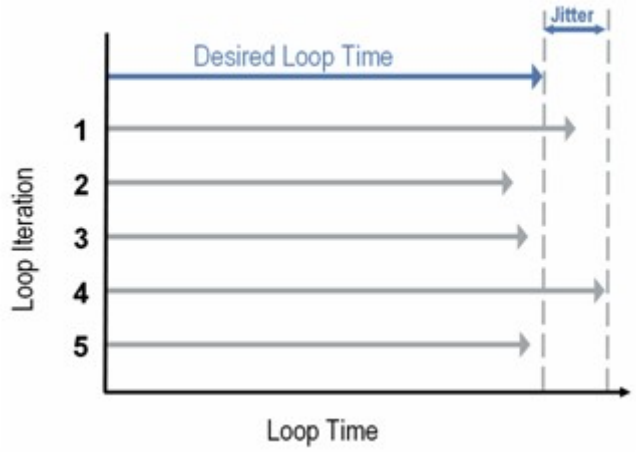

Figure 1: Illustration of the jitter [1]

We can see clearly now, the difference between a real-time OS and a general purpose OS. This difference is the temporal constraint and reliability.

2.2 Why do we need a RTOS?

When the time constraint and the reliability are important, we use a real-time OS. For better understanding of the need of a RTOS, we will explain it with a concrete example.

2.2.1 Example

Suppose that we have an application that executes three tasks: the first one must be executed every 5 milliseconds, we assume that this task performs important calculations for the smooth of the application. The second task, performs the display off different sensors on a screen, this task must be executed every 10 milliseconds to prevent lagging on the screen. This task can be assumed as the less important of the three tasks. And the third and last tasks, send a message on a network (CAN for example) every second precisely, to inform a central unit that it is still active and to send a lot of other information's. Imagine that the central unit activates an alarm if it notices that a module did not send its active message. Thus this third task is the most important one.

Suppose we do not use a RTOS. To implement such an application, it is clear that we will use timers to perform the different tasks. It's a good idea, but on small microcontrollers, there is no priority between different interruptions, so we can ensure that, for example the third task will be well executed every second precisely. Therefore, it's necessary to use a real-time OS, which performs these priorities.

2.3 Multitasking and scheduler

The main feature of an operating system is that it performs multitasking. Time access to the processor is share between all tasks in order to give the illusion to the user that all tasks are running simultaneously. There are different ways to share time access but in this section, I will only explain the cooperative and the preemptive multitasking.

2.3.1 Cooperative multitasking

Cooperative multitasking is the simplest schedulers algorithm. Here, each task give up access to the processor, to allow other tasks taking access to the processor. Each task cooperates with each other to share time access to the processor. And the programmer may not forget the implementation of the giving up of the access to the processor. The reader will perceive, that if a task with an infinite loop does not give up its access time to the processor, the system will be blocked.
On figure 2, task A has access to the processor, while tasks B and C are waiting until task A is finished [5].

Figure 2: Cooperative multitasking between three tasks [5]

The advantage of such a system is that no scheduler is requested and therefore the OS is lighter. The disadvantage is the role of the programmers who may not omit the implementation of the giving up access to the processor and have to reduce the time to access the processor as much as possible.

2.3.2 Preemptive multitasking

In a preemptive multitasking system, it's not anymore the role of each task to give up access to the processor. The operating system has a scheduler whose role is to give access the processor for tasks with higher priority. The scheduler is driven by a regular interrupt named the *clock tick* [6]. On each clock tick, it checks if there is no task with a higher priority that will access to the processor. If so, it will postponed all other tasks until the high priority task is finished. If there are multiple tasks with the same priority level, it depends on the scheduler's algorithm, but generally, the scheduler will give access to the processor every tick for a different task.

On figure 3, we will see how the prioritized preemptive scheduler works.

- Task 1, has a priority level 2, is launched every 4 ticks and needs 1 tick to complete his job.
- Task 2 has a priority level 1, is launched every 3 ticks and needs 3 ticks to complete his job.
- The idle task has a priority level 0. The idle task is always present in a real-time OS, it's the task that runs when there no other tasks that is running.

A short explanation about this timeline:
On tick $t = 1$, the scheduler sees that task 2 must be launched. As task 2 has a higher priority level than the idle task, the scheduler pauses the idle task and launches task 2.

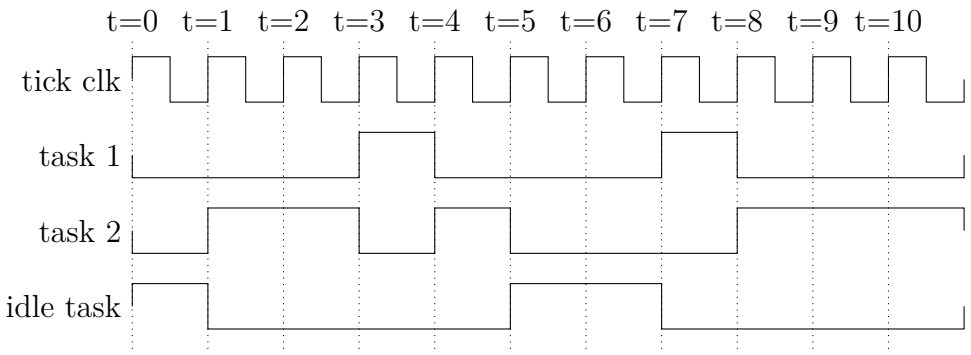

Figure 3: States of different tasks in a prioritized preemptive scheduler algorithm (high = task is running, low = task is paused)

On $t = 3$, task 2 has not yet finished, but as task 3 has a higher priority level, task 2 is paused and task 3 launched. One tick later, task 3 has done its task, the scheduler reactivates task 2.
On $t = 5$, task 2 and 3 do not need access to the processor, the scheduler reactivates the idle task.

Prioritized preemptive multitasking allows a certain level of abstraction in comparison to cooperative multitasking, The only think the programmer has to do is to giving a correct priority for each task so that the program runs correctly. Preemptive multitasking is now used in most of operating systems.

3 Demonstration application

3.1 Introduction

In the demonstration application, we will show the usefulness of a real-time operating system. First we will describe the tools used for our demonstration (Arduino and FreeRTOS). Then, we will explain both; used hardware and software implementation. But first of all, let's introduce our application.

Our application consists in six tasks. The tasks are described below.

- Task A, performs an analog to digital conversion to read the value of a photoresistor.
- Task B, performs an analog to digital conversion to read the value of a thermistor.
- Task C, checks if beams are not interrupted, if yes its sends an error message to the computer within 10 milliseconds.
- Task D, sends all information through USB to a computer.
- Task E, checks if a message is received from the computer, if yes it launches a counter within 5 milliseconds.

- Idle task, where nothing happens.

On table 1, you will find the different properties (priority and time constraint) for the different tasks.

Property / Task	Priority	Time constraint
A	2	Every 10ms
B	2	Every 10ms
C	3	Every 10ms
D	1	Every 1s
E	4	Every 5ms
Idle Task	1	-

Table 1: Priority and time constraint for the different tasks

Task D is the less important task, because it doesn't matter if a delay is introduced in the display, it's the less important operation for the proper functioning of the system. We consider that task A and B as more important because, we assume these measures are used, for example, to control an external system. Task C has a higher priority level because it will represent that something wrong happens with our hypothetical external system. Finally, task E is the most important task because it represents an external interruption done by a user or program.

3.2 Arduino

For this demonstration I have decided to use an Arduino platform, which disregards to configuration and the mounting of a micro-controller which can be sometimes tedious.
I have selected the Arduino platform for its ease of use and the simplicity of the syntax of its programming language Moreover, the Arduino open-source and his community is very active which ensure an ongoing development.

The figure 5 displays a screenshot of the development environment (IDE) of Arduino and the figure 4 retrieves the full description of all pins on the Arduino Nano.

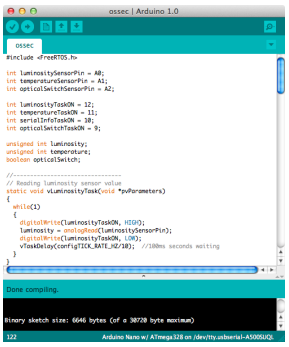

Figure 4: Arduino development environment

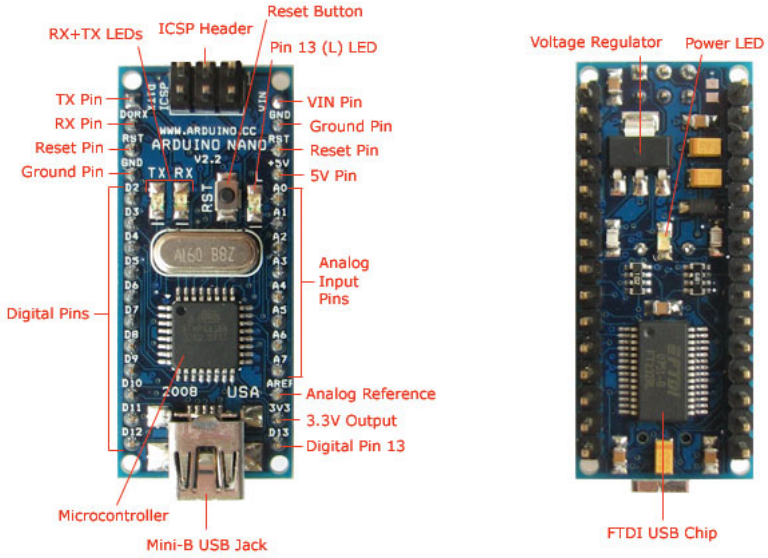

Figure 5: Front and back view of an Arduino Nano board (http : //www.arduino.cc)

3.3 FreeRTOS

The second important part of the demonstration application is the real-time operating system. This was a hard part, because it exists plenty of RTOS for embedded systems. There are also several requirements for choosing a good RTOS, they are listed just below.

- Compatible with ATmega328 chipset.
- Compatible with the Arduino development board.
- Compatible with the last Arduino IDE (version 1.0).
- Support of a prioritized preemptive scheduler
- Light enough to run correctly on an Arduino Nano.

- A good documentation.
- Free.

When all requirements were identified, my choice felt on the FreeRTOS who fulfilled them all perfectly. I found a good integration for the Arduino IDE on

$$http://code.google.com/p/beta-lib/downloads/list$$

3.4 Implementation

Now that we found our tools, we can design our demonstration application. In this section, we will first detail the hardware implementation and finally, describe the used source code.

3.4.1 Hardware

As explain in section 3.1, our system is composed of three sensors.

1. A photoresistor
2. A thermistor
3. An optical switch

On figure 6, we observe a pull-down resistor of $10k\Omega$ to allow the ADC (analog to digital converter) of the Arduino to detect voltage variations. If the pull-down resistor is not present, the ADC will not detect the voltage drop caused by the sensor.
$AREF$ is connected to the $5V$ output pin to ensure that the reference for the ADC is correct. Note that the connection between the Arduino and the computer is not represented on figure 6 and 7.

Now that the schematic is complete, it only remains to mount the circuit on a protoboard. Routing and placement of all the component are shown on figure 7.

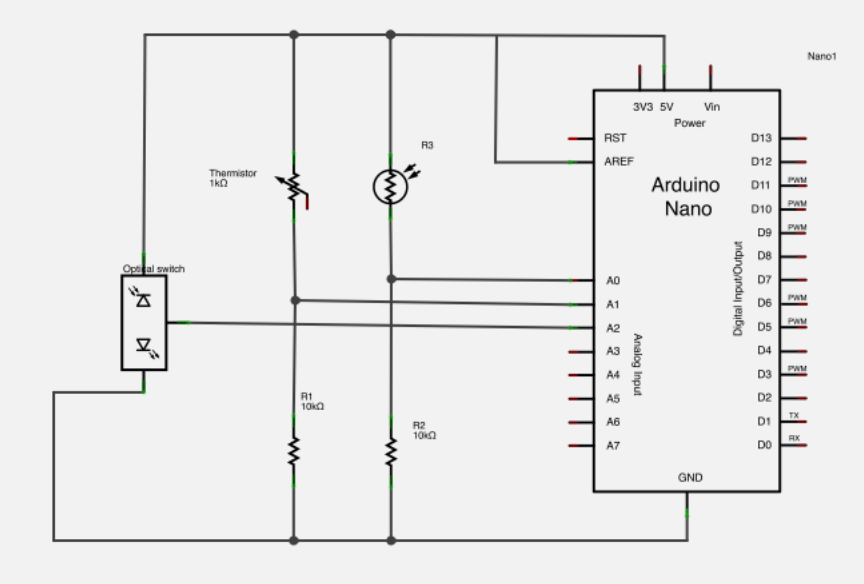

Figure 6: Schematic off the demo application

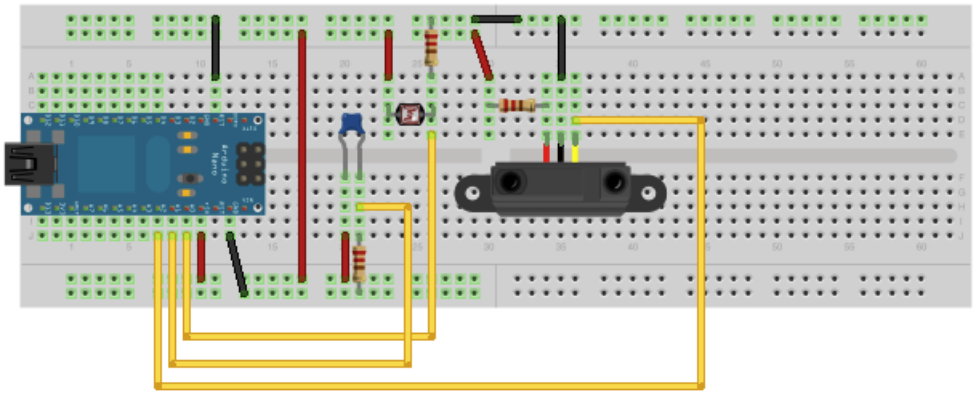

Figure 7: Protoboard off the demo application

3.4.2 Software

Arduino works with at least two functions, which must be implemented. The *setup()* and the *loop()* functions.
The *setup()* function is ran one time directly after a reset. This function serves to initialize each part we need on the Arduino. In our case, the ADC, the serial communication, the FreeRTOS tasks and the input/output pins.
While, the *loop()* is an infinity loop and is ran after the *setup()* function . It's in this function that the whole program is running. But in our case this function is empty.

Working with FreeRTOS on an Arduino platform requires us to adopt another structure for our program. Like all real-time OS, FreeRTOS works with tasks. These tasks must be defined in a function with a well-defined structure, detailed below.

- The initialization code (optional)
- An infinity loop (the task should never return)
- The task application code

Listing 1: Task implementation

```
1 static void vTaskOne(void *pvParameters) {
2
3         //Initialization code
4         while(1) {
5                 //Task application code
6         }
7 }
```

When a task is implemented, you should create this task by calling the *xTaskCreate()* function.
This function takes five parameters (http://www.freertos.org):

- Pointer to the task entry function.
- A descriptive name for the task.
- The size of the task stack.
- Pointer that will be used as the parameter for the task being created.
- The priority at which the task should run.
- A handle by which the created task can be referenced.

This function should be called in the *setup()* function. You will find on listing 2, an example of use of *xTaskCreate()* for *vTaskOne* (listing 1).

Listing 2: Task creation

```
1 xTaskCreate(vTaskOne,
2     (signed portCHAR *) "taskOne",
3     configMINIMAL_STACK_SIZE + 50,
4     NULL,
5     tskIDLE_PRIORITY,
6     NULL);
```

Finally to end our description we will introduce a last concept, named *mutex*. In our application multiple tasks have access to global variables. To avoid that a task wants to change a variable's value while another task is reading this one, we use mutexes.
A mutex, is a king of token. When a task owns the mutex, it is the only one that can access to the variable. Until the task has not given the mutex, other tasks are paused and are waiting until the mutex is free.

Listing 3: Mutex

```
1
2 int globalVariable = 0;
3 xSemaphoreHandle xMutex = NULL;
4
5 void vTaskA (void* pvParameters) {
6
7  while(1){
8   if( xSemaphoreTake( xMutex, portMAX_DELAY == pdTRUE){
9         //Reading globalVariable
10        xSemaphoreGive( xMutex );
11   }
12  }
13 }
14
15 void vTaskB (void* pvParameters) {
16  while(1){
17   if( xSemaphoreTake( xMutex, portMAX_DELAY == pdTRUE){
18        //Changing globalVariable
19        xSemaphoreGive( xMutex );
20   }
21  }
22 }
23
24 setup() {
25  // ...
26  xMutex = xSemaphoreCreateMutex();
27  //...
28 }
```

An example of using of a mutex, is shown on listing 3. First of all, the mutex is created in the *setup()* function. Suppose then that *vTaskA* executes first. Overall it checks if the mutex is available. If yes, *vTaskA* executes its code. Otherwise it waits a delay defined by *portMAX_DELAY* before evaluating the condition again. This piece of code prevents *vTaskA* and *vTaskB* accessing *globalVariable* together.

We are now ready to explain the whole source code of demo application (refer to appendix for the source code of the application). The aims of this part is not to explain line by line the source code but to explain what the tasks, functions do.

setup()

1. Initialization the serial communication (set up the baud rate at 9600 bits/s).
2. Set up the specified digital pins to behave as an output (used for the logical analyzer).

3. Reduce the clock divider of the ADC from 128 to 16 (increase the speed of the conversion).

4. Define the analog reference on the external pin (*AREF*).

5. Create the different tasks.

6. Initialization of the two mutex, used for the temperature and the luminosity variables.

Luminosity and temperature measurements

1. Verify if the mutex is not used.

2. Perform an analog to digital conversion on the correct pin associated to the correct sensor and assign the value of this conversion the correct global variable.

3. Release the mutex.

4. Wait for minimum 10ms, before to run the task again.

Listing 4 shows the measurement of the luminosity sensor.

Listing 4: Reading luminosity sensor value

```
1 static void vLuminosityTask(void *pvParameters)
2 {
3  vTaskSetApplicationTaskTag(NULL, (char(*)(void*)) 1);
4  //Create mutex
5  while(1)
6  {
7   if(xSemaphoreTake(xSemaphoreLuminosity, portMAX_DELAY == pdTRUE))
8   {
9    luminosity = analogRead(luminositySensorPin);
10   xSemaphoreGive(xSemaphoreLuminosity);
11  }
12  vTaskDelay(configTICK_RATE_HZ/100);  //10ms seconds waiting
13 }
14 }
```

Optical switch

1. Verify the optical switch state

2. If ON, sending an error message to the computer and assign value ”true” to the global variable associated to the optical switch.

3. If OFF, assign value ”false” to the global variable.

4. Wait minimum 10ms, before to run the task again.

In this task, I expressly omit to use a mutex to illustrate a task without it.

Listing 5: State checking of the optical switch

```
static void vOpticalSwitchTask(void *pvParameters)
{
 vTaskSetApplicationTaskTag(NULL,  (char(*)(void*))4);
 while(1)
 {
  if(analogRead(opticalSwitchSensorPin) > 50)
  {
   opticalSwitch = true;
   Serial.println("Error");
  }
  else opticalSwitch = false;
  vTaskDelay(configTICK_RATE_HZ/100); //10ms
 }
}
```

Sending information's to the personal computer

1. Verify if the two mutex for the temperature and the luminosity are available.
2. If yes, send via serial all the values of the global variables (luminosity, temperature, optical switch)
3. Wait minimum 1s, before to run the task again.

Listing 6: Sending information to the computer using serial communication

```
static void vSerialInfoTask(void *pvParameters) {
 vTaskSetApplicationTaskTag(NULL, (char(*)(void*))3);
 while(1) {
  if (xSemaphoreTake(xSemaphoreLuminosity, portMAX_DELAY == pdTRUE)
  and xSemaphoreTake(xSemaphoreTemperature,portMAX_DELAY == pdTRUE))
  {
   Serial.print("Luminosity: ");
   Serial.println(luminosity);
   Serial.print("Temperature: ");
   Serial.println(temperature);
   Serial.print("Optical switch: ");
   Serial.println(opticalSwitch);
   xSemaphoreGive(xSemaphoreLuminosity);
   xSemaphoreGive(xSemaphoreTemperature);
  }
  vTaskDelay(configTICK_RATE_HZ);
 }
}
```

Receiving information from the personal computer

1. Check if serial communication if available.
2. Check if the stop message is received (character *s*).
3. Freeze the application for 3s.
4. Wait for 5ms, before to run the task again.

Listing 7: Reading informations from the serial communication

```
1 static void vSerialMsgCheck(void *pvParameters)
2 {
3  vTaskSetApplicationTaskTag(NULL, (char(*)(void*))5);
4  while(1) {
5   if (Serial.available() > 0)
6   {
7    if ( Serial.read() == 's')
8    {
9     delay(3000); //Force waiting 1 second
10   }
11  }
12  vTaskDelay(configTICK_RATE_HZ/200); //5ms
13 }
14 }
```

3.5 Results

Now that the demonstration application is designed, we have to check if it works as we expect. To do so we will use a logic analyzer and the trace features from FreeRTOS to collect information's about running tasks (it will set a specified digital pin for each task on a logic level 1 when this specific task is running).

The data given by the logic analyzer are shown on figures 8 and 9. We observe that the results are those we expected. Figure 8, task D has the mutex and until the task give the mutex back other tasks even with a higher priority are paused. On the other side, when we do not use mutex's, task D is paused until task C has finished. This last effect is shown on time line on figure 9, and visualized on figure 10. Information's messages from task D is paused, the error message is send, and then the end of the message arrives. These priority levels are also observed when the personal computer sends to the Arduino the an 's'. All the system is paused during 3 seconds.

Fortunately, we see in the results that the system design does what it was expected for. It's fits perfectly with the theory of a prioritized preemptive scheduler.

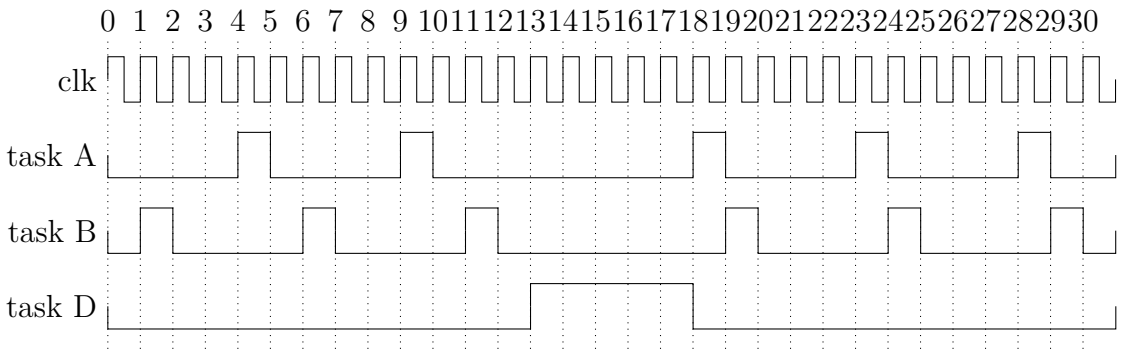

Figure 8: Effect of the use of mutex's

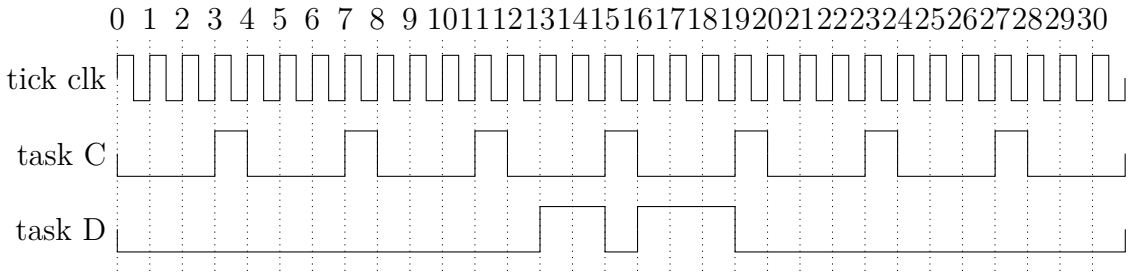

Figure 9: Scheduler pausing lower priority task D to allows task C to run.

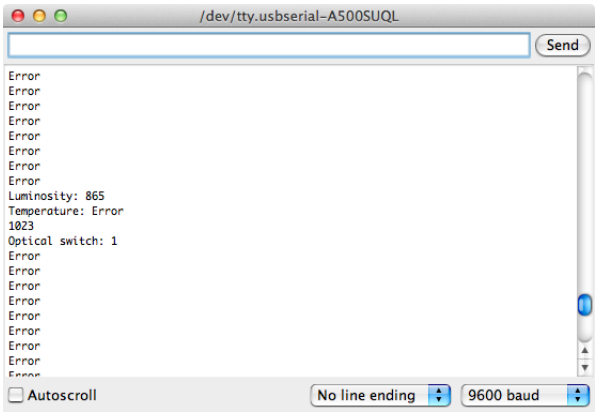

Figure 10: Visualisation of the different priority level for different tasks.

4 Conclusion

In this report we have seen on introduction on how works a real-time operating systems and an overview of FreeRTOS and the Arduino development platform. It allows to better understand the usefulness of real-time OS. The aim of this report is not to enter in all details of the scheduler, but well to understand how it works. Especially to understand in which case an real-time OS is needed.

List of Figures

1 Illustration of the jitter [1] 3
2 Cooperative multitasking between three tasks [5] 4
3 States of different tasks in a prioritized preemptive scheduler algorithm (high = task is running, low = task is paused) 5
4 Arduino development environment 7
5 Front and back view of an Arduino Nano board (http : //www.arduino.cc) 7
6 Schematic off the demo application 9
7 Protoboard off the demo application 9
8 Effect of the use of mutex's 15
9 Scheduler pausing lower priority task D to allows task C to run. 15
10 Visualisation of the different priority level for different tasks. 15

List of Tables

1 Priority and time constraint for the different tasks 6

References

[1] N. Instruments, "What is a real-time operating system (rtos)." http://zone.ni.com/devzone/cda/tut/p/id/3938, January 2010.

[2] S. Rivoir, "Real time operating system." http://www.stanford.edu/ skrufi/r-tospres.ppt, November 2002.

[3] P. Mathys, *ELEC-H410 - Real-time systems*. Université Libre de Bruxelles, 2012.

[4] Wikipedia, "Real time operating system." http://en.wikipedia.org/wiki/rtos, April 2012.

[5] K. Hyder and B. Perrin, *Embedded Systems Design Using the Rabbit 3000 Microprocessor: Interfacing, Networking, and Application Design*. Embedded Technology Series, Newnes, 2004.

[6] T. Wilmshurst, *Designing Embedded Systems With PIC Microcontrollers: Principles and Applications*. Elsevier Science, 2009.

[7] FreeRTOS, "Freertos implementation." http://www.freertos.org/implementation/index.html.

[8] D. Ibrahim, *Advanced PIC Microcontroller Projects in C: From USB to RTOS with the PIC18F Series*. Electronics & Electrical, Newnes/Elsevier, 2008.

[9] Arduino, "Arduino official website." http://www.arduino.cc, February 2012.

A Source code

Listing 8: Full source code

```
1 #include <FreeRTOS.h>
2
3 #define traceTASK_SWITCHED_IN() vSetDigitalOutput( (int)
4                     pxCurrentTCB->pxTaskTag)
5
6 int luminositySensorPin = A0;
7 int temperatureSensorPin = A1;
8 int opticalSwitchSensorPin = A2;
9
10 int luminosityTaskON = 12;
11 int temperatureTaskON = 11;
12 int serialInfoTaskON = 10;
13 int opticalSwitchTaskON = 9;
14 int serialMsgTaskON = 8;
15 int idleTaskON = 7;
16
17 unsigned int luminosity;
18 unsigned int temperature;
19 boolean opticalSwitch;
20
21 xSemaphoreHandle xSemaphoreLuminosity;
22 xSemaphoreHandle xSemaphoreTemperature;
23
24 //-------------------------------------
25 // Reading luminosity sensor value
26 static void vLuminosityTask(void *pvParameters)
27 {
28   vTaskSetApplicationTaskTag(NULL, (char(*)(void*)) 1);
29   //Create mutex
30   while(1)
31   {
32     if (xSemaphoreTake(xSemaphoreLuminosity, portMAX_DELAY == pdTRUE))
33     {
34       luminosity = analogRead(luminositySensorPin);
35       xSemaphoreGive(xSemaphoreLuminosity);
36     }
37     vTaskDelay(configTICK_RATE_HZ/100);  //10ms seconds waiting
38   }
39 }
40
41 //-------------------------------------
42 // Reading luminosity sensor value
43 static void vTemperatureTask(void *pvParameters)
44 {
```

```
45    vTaskSetApplicationTaskTag(NULL, (char(*)(void*)) 2);
46    while(1)
47    {
48      if (xSemaphoreTake(xSemaphoreTemperature, portMAX_DELAY == pdTRUE))
49      {
50        temperature = analogRead(temperatureSensorPin);
51        xSemaphoreGive(xSemaphoreTemperature);
52      }
53      vTaskDelay(configTICK_RATE_HZ/100);   //10ms
54    }
55 }
56
57 //-------------------------------------
58 // Reading optocaptor sensor value
59 static void vOpticalSwitchTask(void *pvParameters)
60 {
61    vTaskSetApplicationTaskTag(NULL,  (char(*)(void*))4);
62    unsigned int temp;
63    while(1)
64    {
65      if(analogRead(opticalSwitchSensorPin) > 50)
66      {
67        opticalSwitch = true;
68        Serial.println("Error");
69      }
70      else opticalSwitch = false;
71      vTaskDelay(configTICK_RATE_HZ/100); //10ms
72    }
73 }
74
75 //-------------------------------------------------------------------------
76 // Send informations via Serial interface
77 static void vSerialInfoTask(void *pvParameters) {
78    vTaskSetApplicationTaskTag(NULL, (char(*)(void*))3);
79    while(1) {
80      if (xSemaphoreTake(xSemaphoreLuminosity, portMAX_DELAY == pdTRUE) an
81      {
82        Serial.print("Luminosity: ");
83        Serial.println(luminosity);
84        Serial.print("Temperature: ");
85        Serial.println(temperature);
86        Serial.print("Optical switch: ");
87        Serial.println(opticalSwitch);
88        xSemaphoreGive(xSemaphoreLuminosity);
89        xSemaphoreGive(xSemaphoreTemperature);
90      }
91      vTaskDelay(configTICK_RATE_HZ); //1 secondes waiting
```

```
92    }
93 }
94
95 //-------------------------------------------
96
97 static void vSerialMsgCheck(void *pvParameters)
98 {
99    vTaskSetApplicationTaskTag(NULL, (char(*)(void*))5);
100   while(1) {
101     if (Serial.available() > 0)
102     {
103       int receivedByte = Serial.read();
104       if (receivedByte == 's')
105       {
106         delay(3000); //Force waiting 1 second
107       }
108     }
109     vTaskDelay(configTICK_RATE_HZ/200); //5ms
110   }
111 }
112
113 void vIdleTask(void *pvParameters) {
114   vTaskSetApplicationTaskTag(NULL, (char(*)(void*))6);
115   while(1)
116   {}
117 }
118
119 //-------------------------------------------------------------------------
120 void setup() {
121   //Initialization of the serial communication with 9600kbits as baud ra
122   Serial.begin(9600);
123
124   pinMode(luminosityTaskON, OUTPUT);
125   pinMode(temperatureTaskON, OUTPUT);
126   pinMode(serialInfoTaskON, OUTPUT);
127   pinMode(opticalSwitchTaskON, OUTPUT);
128   pinMode(serialMsgTaskON, OUTPUT);
129   pinMode(idleTaskON, OUTPUT);
130
131   //Configuring HIGH SPEED
132   bitClear(ADCSRA, ADPS0);
133   bitClear(ADCSRA, ADPS1);
134   bitSet(ADCSRA, ADPS2);
135   analogReference(EXTERNAL);
136
137   // create luminosity measurement task
138   xTaskCreate(vLuminosityTask,
```

```
139     (signed portCHAR *)"luminosityTask",
140     configMINIMAL_STACK_SIZE + 50,
141     NULL,
142     tskIDLE_PRIORITY + 2,
143     NULL);
144
145   // create temperature measurement task
146   xTaskCreate(vTemperatureTask,
147     (signed portCHAR *)"temperatureTask",
148     configMINIMAL_STACK_SIZE + 50,
149     NULL,
150     tskIDLE_PRIORITY + 2,
151     NULL);
152
153    xTaskCreate(vOpticalSwitchTask,
154     (signed portCHAR *)"opticalSwitchTask",
155     configMINIMAL_STACK_SIZE + 50,
156     NULL,
157     tskIDLE_PRIORITY + 3,
158     NULL);
159
160   xTaskCreate(vSerialMsgCheck,
161     (signed portCHAR *)"serialMsgCheckTask",
162     configMINIMAL_STACK_SIZE + 50,
163     NULL,
164     tskIDLE_PRIORITY + 4,
165     NULL);
166
167   xTaskCreate(vSerialInfoTask,
168     (signed portCHAR *)"serialTask",
169     configMINIMAL_STACK_SIZE + 50,
170     NULL,
171     tskIDLE_PRIORITY + 1,
172     NULL);
173
174   xTaskCreate(vIdleTask,
175     (signed portCHAR *)"idleTask",
176     configMINIMAL_STACK_SIZE + 50,
177     NULL,
178     tskIDLE_PRIORITY,
179     NULL);
180
181   xSemaphoreLuminosity = xSemaphoreCreateMutex();
182   xSemaphoreTemperature = xSemaphoreCreateMutex();
183
184   // start FreeRTOS
185   vTaskStartScheduler();
```

```
186
187   Serial.println("Die");
188   while(1);
189 }
190
191 void vSetDigitalOutput(int task)
192 {
193   digitalWrite(temperatureTaskON, LOW);
194   digitalWrite(serialInfoTaskON, LOW);
195   digitalWrite(opticalSwitchTaskON, LOW);
196   digitalWrite(serialMsgTaskON, LOW);
197   digitalWrite(idleTaskON, LOW);
198   digitalWrite(temperatureTaskON, LOW);
199   switch (task){
200     case 1:
201       digitalWrite(luminosityTaskON, HIGH);
202     break;
203     case 2:
204       digitalWrite(temperatureTaskON, HIGH);
205     break;
206     case 3:
207       digitalWrite(serialInfoTaskON, HIGH);
208     break;
209     case 4:
210       digitalWrite(opticalSwitchTaskON, HIGH);
211     break;
212     case 5:
213       digitalWrite(serialMsgTaskON, HIGH);
214     break;
215     case 6:
216       digitalWrite(idleTaskON, HIGH);
217     break;
218     default:
219     break;
220
221   }
222 }
223
224 void loop()
225 {}
```