Table of Contents

Introduction
Multiprogramming using an ATMEL Microcontroller

In general, real-time control and response actions will be critical for many embedded systems when timing constraints must be met. For a great number of applications such as industrial management (Industry 4.0), medical assistance, mission-critical control, vehicular automation, among others, complying with real-time requirements may be a fundamental demand when implementing embedded systems. In this complex scenario, real-time multi-tasking in microcontrollers may play an important role, pushing us to better understand their main development challenges and major designing issues.

Real-time tasks are expected to be *time-deterministic,* meaning that the response time to any event must respect a defined value (threshold), usually determined at project time. In order to help when accomplishing that goal, Real Time Operating Systems (RTOS) have to be employed to support the concurrent operation of multiple tasks, particularly when embedded applications have tight timing requirements and high reliability demands. Since a RTOS may help to properly implement multi-tasking according to all mentioned development constraints, the main goal of this practical activity guide is to encourage students to exploit real-time multi-tasking resources when using a RTOS on an affordable microcontroller, taking as reference the programming on an Arduino UNO board with the support of the FreeRTOS libraries.

This laboratory work will be divided into 3 sessions with the following general objectives:

- ***Part A -*** *ATMEL microcontroller (ATmega328,* ***I/Os****,* ***Interruptions*** *and* ***Timers****).*
- ***Part B -*** *Basic concepts of FreeRTOS:* ***Tasks, Preemption*** *and* ***Task Control****.*
- ***Part C -*** *Advanced concepts of FreeRTOS:* ***Race conditions*** *and* ***Mutual Exclusion****.*

Some complementary readings:

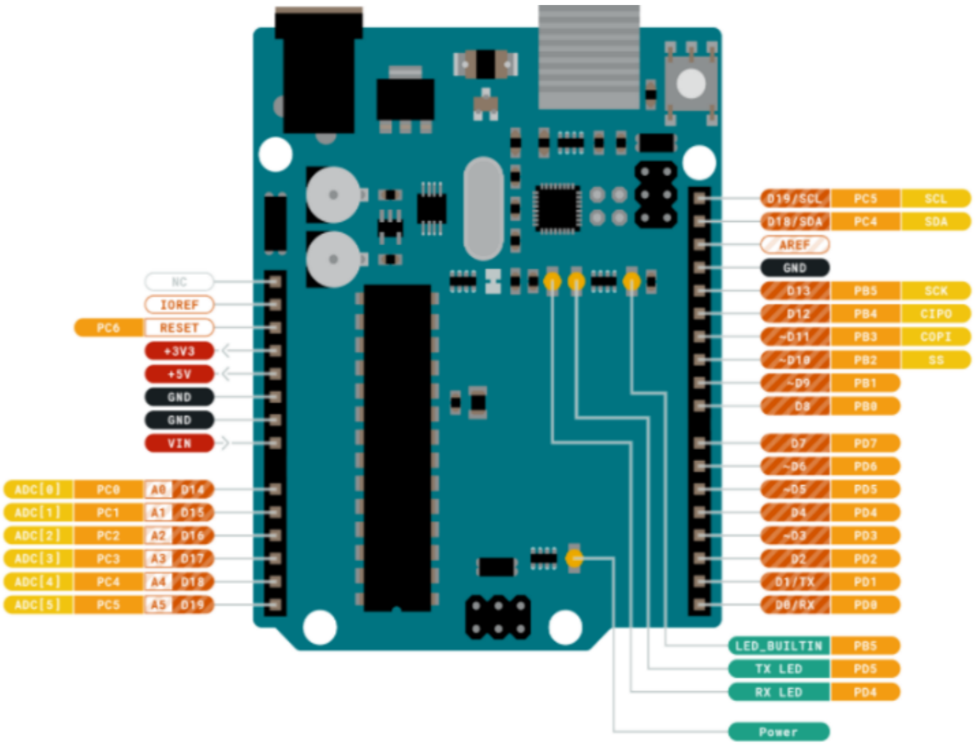
- Annex - set up your machine
- Microcontroller: a single integrated circuit chip for a complete computer system (a CPU, memory, a clock oscillator and I/Os). For ATmega328: read pages 25 to 36 of the datasheet.
- Slides about "*Microcontroller: Arduino board*"
- Datasheet "Multifunction Shield".
- FreeRTOS:
  - Online documentation (as needed):
    https://www.freertos.org/Documentation/RTOS_book.html
    https://www.freertos.org/FreeRTOS-quick-start-guide.html

  - Code (open source): *https://github.com/FreeRTOS*
- Chapter 28 (Locks - first 2 pages), 31 (Semaphores - first 10 pages) and 32 (Concurrency Bugs - optional) of the book *Operating Systems: Three Easy Pieces*, Remzi H. and Andrea C. (University of Wisconsin-Madison). We will revisit these topics later on.
  - *https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf*
  - *https://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf*
  - *https://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf*

Part A - ATMEL microcontroller

Introduction to Arduino Uno:

The Arduino Uno board resorts to a microcontroller ATmega328P (from the AVR Family). The basic features of this board and the pinout are depicted below.



Interfacing			Hardware overview	
I/Os	Built-in LED Pin	13	Microcontroller	ATmega328P (8 bits)
	Digital I/O Pins	14	Voltage	5V
	Analog input pins	6	Clock speed	16MHz
	PWM pins	6	Flash memory	32KB
Communication	UART	Yes	SRAM	2KB
	I2C	Yes	EEPROM	1KB
	SPI	Yes	Weight	25g

Figure 1 - Arduino UNO hardware summary.

Introduction to the AVR Library:

One of the main challenges when programming an embedded system (and particularly a hardware development platform such as the Arduino) is to find appropriate programming libraries that facilitate the interaction with the hardware resources. In fact, there are some available options concerning both languages (e.g. C and Python) and libraries, with different particularities when implementing concurrency. Since the language C is the reference for this course, this section presents one of the libraries when pursuing Arduino-based embedded multi-tasking.

The AVR GCC Library tries to keep the rules for its embedded C (for AVR controllers) close to ANSI C. This library defines a set of macros to interact directly with the hardware, such as to access hardware registers.

An initial example when using this library is presented as follows:

AVR Program 1

```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
    // initialization
    DDRB |= (1<<5);   // make the LED pin (PB5) an output

    // enter endless recurrent program (task)
    while(1) {
       // switch on LED, wait, switch off LED, wait
       PORTB |= (1<<5); // forces 1 in bit 5 (LED ON)
       _delay_ms(200);  // wait 200ms
       PORTB &= ~(1<<5); // forces 0 in bit 5 (LED OFF)
      _delay_ms(200);  // wait 200ms
    }
}
```

The registers are treated as variables, e.g., ‘DDRB=0xFF’ using the hexadecimal format in C. The most important headers to include in a project are: ‘avr/io.h’, ‘util/delay.h’ and ‘avr/interrupt.h’. These libraries provide direct access to the microcontroller functionalities/hardware, such as: Digital I/O, ADCs, UARTs, Timers, Interruptions, etc. However, they have severe limitations in terms of concurrency of multiple tasks and portability to other platforms. Hence, the programmer needs to have a complete knowledge about the details of the microcontroller, i.e., this way of programming has no virtualization and consequently, a very low level of abstraction.

The AVR library can be used directly in the Arduino IDE. Test the example ***AVR Program 1*** using the computer that is available in your laboratory (alternatively, you can simply analyze the code).

Although the direct use of the AVR library can give enhanced control of the operation of the microcontrollers, it may be tricky and prone to errors for more complex applications. This way, we suggest that the standard Arduino C library be used instead.

Introduction to the Arduino C Library:

The Arduino C library provides a hardware abstraction that is more user-friendly when compared to the AVR library, being largely adopted to develop embedded systems based on Arduino boards. However, there is a certain price to pay, since the programmer loses a little of the direct hardware control (this can be a problem when we need to work at the limits of the CPU, e.g., to carry out operations at very precise times, within a few microseconds). Nevertheless, when properly implemented using the standard Arduino C library, high performance is still achieved while the implemented code becomes more readable and easier to be maintained and upgraded.

The documentation about the use of the Arduino C is vast, with many examples and code troubleshooting in books, blogs and even video materials in popular platforms like YouTube.

An initial example of using this library is presented as follows, which is a re-implementation of the example 1 based on the AVR library:

Arduino Program 1

```
const int ledPin = LED_BUILTIN; // define which LED to use
const long interval = 200; // define interval to toggle LED in ms

// initialization
void setup() {
    pinMode (ledPin, OUTPUT);   // set the ledPin as output
}

// run
void loop() {
    // toggles ledPin every cycle, cycle takes ~2*interval
    digitalWrite(ledPin, HIGH); // switches on ledPin
    delay(interval);          // wait “interval” time
    digitalWrite(ledPin, LOW); // switches off ledPin
    delay(interval);          // wait “interval” time
}
```

By re-implementing AVR Program 1 using the Arduino C library, we can see the enhanced level of abstraction that this library offers, hiding several hardware-related details.

In the following example, a serial interface is used to receive an input from the user, allowing that the delay time is changed during the execution.

Arduino Program 2

```
const int ledPin = LED_BUILTIN; // define which LED to use
long interval = 200; // define interval to toggle LED

// initialization
void setup() {
    pinMode (ledPin, OUTPUT);   // set the ledPin as output

    Serial.begin(9600);
}
```

```
// run
void loop() {

    // toggles ledPin every cycle, interval provided by the user
    digitalWrite(ledPin, HIGH); // switches on ledPin
    delay(interval);         // wait “interval” time
    digitalWrite(ledPin, LOW); // switches off ledPin
    delay(interval);         // wait “interval” time

    interval = Serial.parseInt();
}
```

The variable *interval* is not a constant anymore, receiving the input from the user through the serial interface. The biggest problem of this approach is that the interaction with the user **block** the execution of the program until a proper input is provided. Since there is a simple concurrency in this program (LED blinking and user interaction), the way it is implemented prevent that the developed system operates with concurrent actions.

Initial steps in concurrency:

In a single-core system as the Arduino Uno board, concurrency is achieved by giving to the users the feeling that more than one thing (task) is being executed at the same time. As discussed before, blocking the standard execution flow must be avoided, which requires different strategies when tackling this problem. Actually, there are three popular ways to provide different levels of concurrency in an embedded system: Finite State Machines (FSMs), Interruptions and RTOS-based preemptions.

The use of state machines is based on the fact that a single execution flow can allow the execution of different pieces of code according to the accounting of timestamps. This way, different state machines may be executed at once, with a particular scope of execution, although blocking conditions still need to be avoided.

The following example allows two LEDs to blink at different rates exploiting a very simple implementation of FSM.

FSM Program 1

```
unsigned long initialTimeLED1 = 0;
unsigned long initialTimeLED2 = 0;

long intervalLED1 = 1000;  // 1 second for SM1
long intervalLED2 = 2500;  // 2.5 seconds for SM2

const int ledPin1 = 6;
const int ledPin2 = 7;

int stateLED1 = HIGH;
int stateLED2 = HIGH;

// initialization
void setup() {
    pinMode (ledPin1, OUTPUT);   // set the digitial Pin 6 as output
    pinMode (ledPin2, OUTPUT);   // set the digitial Pin 7 as output

    initialTimeLED1 = millis();
    initialTimeLED2 = millis();
}
```

```
// run
void loop() {

  //State machine 1
  if ((millis() - initialTimeLED1) < intervalLED1)
  {
      digitalWrite(ledPin1, stateLED1);
  }
  else
  {
     stateLED1 = !stateLED1;
     initialTimeLED1 = millis();
  }

  //State machine 2
  if ((millis() - initialTimeLED2) < intervalLED2)
  {
      digitalWrite(ledPin2, stateLED2);
  }
  else
  {
     stateLED2 = !stateLED2;
     initialTimeLED2 = millis();
  }
}
```

Concurrency can also be implemented with the use of hardware interrupts. These are signals that cause the CPU to stop executing its program instructions and jump to a specified routine, called the interrupt service routine (ISR). At the end of the ISR, the CPU continues executing its instructions starting from where it stopped. In the following example we will use interrupts generated periodically by a Timer unit.

Therefore, an ISR can be used to execute one or more tasks that run concurrently with the main program. Implicitly, tasks ***task_LED1*** and ***task_LED2*** are always invoked in sequence by the ISR and they will run concurrently with the main program (note that the main program is locked in an infinite loop that does nothing but spin on itself, being interrupted by the periodic timer interrupts - let's call them "ticks"). Each task uses a global variable to count ticks and do something just when the count reaches a predefined value, i.e., after an integer number of "ticks".

AVR Program 2

```
#include <avr/io.h>
#include <util/delay.h>
// HW Timer2 CTC mode
#define T2TOP 250
// Count of SW timers
#define TIME1 125
#define TIME2 63
// LEDs that will be used
#define LED1 PB5 // pin 13
#define LED2 PB2 // pin 10

// Global variables…

// variables for SW timers (tick counters)
uint8_t time1 = TIME1;
uint8_t time2 = TIME2;
```

```
/////////////////////////
// program timer2 in CTC mode
// generate interrupt every 4ms
void tc2_init(void) {
  TCCR2B = 0;       // stop TC2
  TIFR2 |= (7<<TOV2);  // clear pending interrupt
  TCCR2A = 2;       // mode CTC
  TCNT2 = 0;        // BOTTOM value
  OCR2A = T2TOP;    // TOP value
  TIMSK2 |= (1<<OCIE2A);  // enable COMPA interrupt
  TCCR2B = 6;       // start TC2  (TP=256)
}

////////////////
// task_LED1
// toggles LED1 every TIME1 ticks (SW timer 1)
void task_LED1(void) {
  if(time1)
    time1--;  // if time1 not 0, decrement
  else {
    // do here the task code, to run every TIME1 ticks
    PORTB ^= (1<<LED1);  // toggles LED1 in PORTB

    time1 = TIME1;      // reset to SW timer 1
  }
}

////////////////
// task_LED2
// toggles LED2 every TIME2 ticks (SW timer 2)
void task_LED2(void) {
    if(time2)
        time2--;  // if time2 not 0, decrement
    else {
    // do here the task code, to run every TIME2 ticks
        PORTB ^= (1<<LED2);  // toggles LED2 in PORTB
        time2 = TIME2;      // reset to SW timer 2
  }
}

// Timer 2 ISR for CTC mode
// every interrupt invoke existing tasks
ISR(TIMER2_COMPA_vect) {
    task_LED1();
    task_LED2();
}

// main
int main(void) {
// initialization
    DDRB |= (1<<LED1) | (1<<LED2); // LED1, LED2 pins -> outputs
    tc2_init();  // initialize timer2
    sei();  // enable interrupts

    // enter endless loop (processing done in the tasks)
    // this loop can be used to run code, too, but remember
    // it is recurrently interrupted by the tasks
    while(1) { };
}
```

This is a simple way of controlling the periodic execution of a task, in which the actual control (when to execute) is defined by the task itself, based on a timebase defined by the Timer tick period. The tasks scheduler is implemented inside the ISR invoking tasks sequentially. Thus, one task must finish before another one starts.

This technique already virtualizes the CPU in a very simple way, allowing the execution of tasks concurrently with the main program. However, it is still very limited in the tasks control and with a rather low abstraction level (the programmer still has to know the hardware registers and handle the ISR routine directly). A graphical representation of the AVR Program 2 can be seen in Figure 2, but note that the two tasks never interrupt each other.

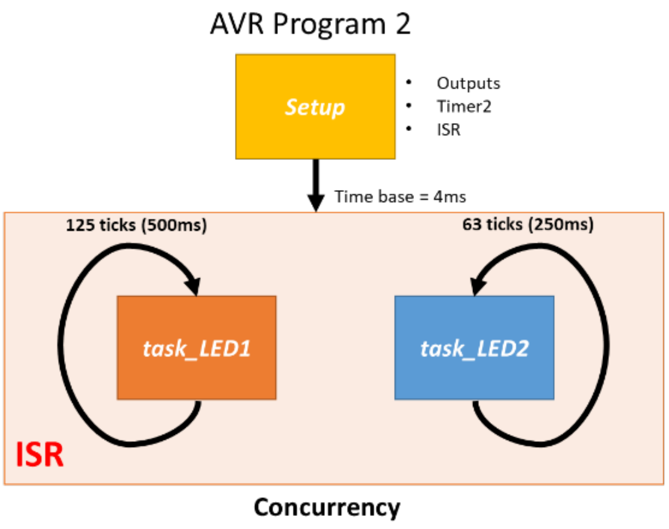


Figure 2 - Graphical representation of program 2. The prescaler for Timer 2 is set to 256, which gives a sample time of 16ms for Timer 2. Since the interruption is generated when Timer 2 reaches the value of 250, the ISR period is 4ms. Then, for task 1 (task 2), the LED state switches after 125 (64) ISR periods, that is, 500ms (250ms).

The Arduino multifunction shield

Some interactions with the users through inputs and outputs can be facilitated by the adoption of the Arduino multifunction shield. It is a very affordable shield for the Arduino Uno board, having a 7-segment display, 4 LEDs, a 10K potentiometer, 3 push buttons, a buzzer, a servo interface, among other useful resources.

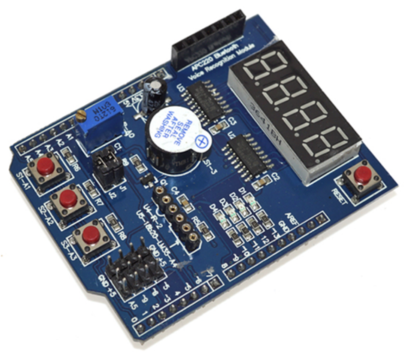


Figure 3 – The Arduino multifuction shield that will be used (https://www.ptrobotics.com/shields-varios/6243-multi-function-shield-for-arduino.html).

The resources of the Arduino multifunction shield can be easily accessed through its library, which can be directly installed using the Arduino IDE library manager (see Annex). The resources of the shield will be accessed when importing that library:

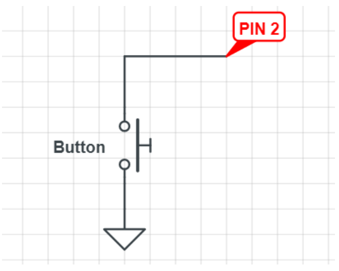
```
#include <TimerOne.h>
#include <Wire.h>
#include <MultiFuncShield.h>

void setup()
{
  Timer1.initialize();
  MFS.initialize(&Timer1);    // initialize multi-function shield library

  // Beep control is performed in the background, i.e. beep() is non blocking.
  // short beep for 200 milliseconds
  MFS.beep();

  //Light all LEDs
  MFS.writeLeds(LED_ALL, ON);

  for (int i = 3; i >= 1; i--)
  {
    //Show the result the Display
    MFS.write(i);
    delay(1000);
  }
  MFS.write("");
  MFS.writeLeds(LED_ALL, OFF);

  // 4 short beeps, repeated 3 times.
  MFS.beep(5,    // beep for 50 milliseconds
           5,    // silent for 50 milliseconds
           4,    // repeat above cycle 4 times
           3,    // loop 3 times
           50    // wait 500 milliseconds between loop
           );
}

void loop() {
  //Read the buttons
  byte btn = MFS.getButton();

  //Present the results
  if (btn == BUTTON_1_PRESSED)
  {
    MFS.write("1");
    MFS.writeLeds(LED_1, ON);
    MFS.writeLeds(LED_2, OFF);
    MFS.writeLeds(LED_3, OFF);
  }
  if (btn == BUTTON_2_PRESSED)
  {
    MFS.write("2");
    MFS.writeLeds(LED_1, OFF);
    MFS.writeLeds(LED_2, ON);
    MFS.writeLeds(LED_3, OFF);
  }
  if (btn == BUTTON_3_PRESSED)
  {
    MFS.write("3");
    MFS.writeLeds(LED_1, OFF);
    MFS.writeLeds(LED_2, OFF);
    MFS.writeLeds(LED_3, ON);
  }
}
```

Exercise 1 – Non-blocking LED switching and buttons control

The FSM Program 1 blinks two LEDs at different frequencies using for that the concept of finite state machines, without using the ***delay*** function to force the CPU to wait (making it “blocked”). We say “blocked” because this function does a “busy-wait” cycle, in which the CPU is kept running in a cycle to spend time, thus “blocked”. For that non-blocking program, implement an enhanced code that take as input two push buttons (PB1 and PB2). While PB1 is kept pressed, LED1 stops its normal operation, and the corresponding function is expected for PB2 regarding LED2. The LEDs return to the normal blinking operation when the corresponding buttons are released.

In this exercise, an important design issue is to decide if the LED blinking procedure is **resumed** (the FSM time variables are restored) or if the blinking cycle is **reset**.

Exercise 2 – Using physical interruptions to emulate concurrency



Working with interruptions is also possible with the Arduino C library. Change the code in Exercise 1 to make the buttons operate through interruptions (there are 2 in Arduino Uno), one for each button, using the Arduino resources for that. The 2 buttons will activate or deactivate the blinking operation of the corresponding LEDs through an ISR (*Press -> Activate… Press -> Deactivate*), which is a different behavior for the buttons when compared with Exercise 1.

Note: consider the **bouncing effect** of the buttons and discuss possible solutions.

Exercise 3 – Using time interruptions to emulate concurrency

One possibility when concurrently handling multiple "functionalities" is defining a time-based interruption procedure that will trigger the execution of a function at a constant frequency (such as in AVR Program 2). When done in Arduino programs, it could give a feeling of concurrent programming, although there are no guarantees about the required processing time after an interruption is triggered.

In this exercise, use the library **TimerOne.h**, considering this following example:

```
#include <TimerOne.h>

//Function called by the Timer as an interruption
void blinkLED1(void);

void setup()
{
  Timer1.initialize(1000000); //in microseconds
  Timer1.attachInterrupt(blinkLED1);
}
```

Note: the Arduino has different built-in timers that can be accessed in different ways.

Part B - Concepts of FreeRTOS

Introduction to the FreeRTOS Library:

FreeRTOS is a very popular open-source Real-Time Operating System (RTOS) that has been largely used to provide efficient multitasking capabilities to resource-constrained microcontroller-based systems. Considering the Arduino AVR platform, FreeRTOS has been a reliable option as a RTOS, combining both processing efficiency with optimized use of the available resources.

Let's create two tasks using the FreeRTOS and Arduino IDE:

- ***TaskBlink1*** that toggles LED D1 with a frequency of 1Hz;
- ***TaskPrint*** that counts seconds and sends the value over the Serial port (UART at 9600 baud rate) at the same frequency of 1Hz.

Both tasks should run concurrently and indefinitely, until the Arduino board is forcibly reset or switched off. The FreeRTOS makes it possible to avoid a sequential execution of routines which is crucial for running multiple tasks at the same time. In fact, for the users, there is a general perception that both tasks are indeed on a concurrent operation, although only a single task is being handled by the microcontroller at a given time. Since a single-core microcontroller is being adopted (ATmega328P), the FreeRTOS will perform the required preemption procedures in a transparent way, facilitating the programming of embedded systems.

Figure 4 presents a schema when adopting a RTOS-based programming approach.

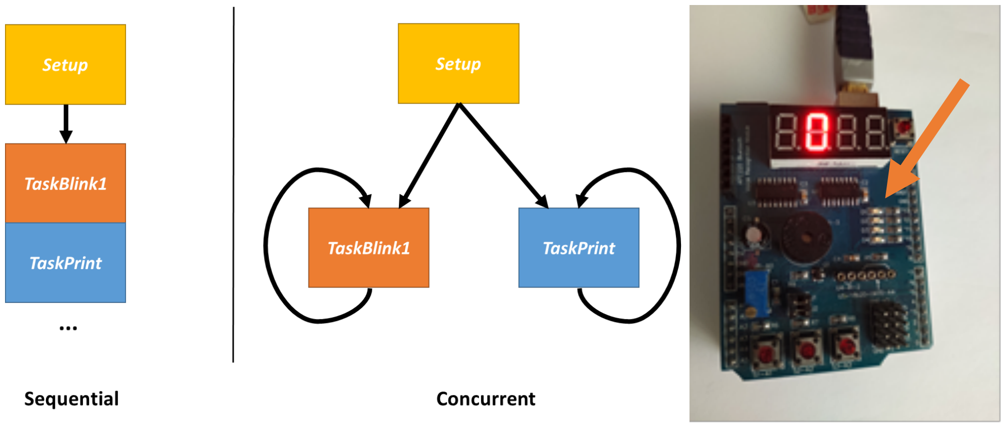


Figure 4 - Execution of two routines (serial communication and LED Blink) using a sequential or concurrent execution.

Let's start with the basics of the FreeRTOS programming. First, import the Arduino_FreeRTOS and MultiFuncShield headers and create the function prototype for each task that will be executed.

```
#include <Arduino_FreeRTOS.h>
#include <TimerOne.h>
#include <MultiFuncShield.h>

void taskBlink1 (void *pvParameters);
void taskPrint (void *pvParameters);
```

Then, initialize the serial communication (9600 baud rate), and create two tasks within the **setup**() function. For that, one should use the FreeRTOS API, particularly the ***xTaskCreate***[1] and ***vTaskStartScheduler***[2] functions.

The following example presents how the intended tasks can be created.

```
void setup() {
  //Initialize the shield
  Timer1.initialize();
  MFS.initialize(&Timer1);

  //Initialize the Serial interface
  Serial.begin(9600);

  xTaskCreate (taskBlink1, //Function
               "Led1",      //Description
               128,         //Stack "size"
               NULL,        //Parameters for the function
               1,           //Priority
               NULL);       //Task handler

  xTaskCreate (taskPrint, "Print", 128, NULL, 1, NULL);

  vTaskStartScheduler();
}
```

The next step is to implement the functions ***TaskBlink1*** and ***TaskPrint***, which will be done using the same resources of conventional Arduino programming. The exception is the "pause" by the ***vTaskDelay***[3] function, that must be used instead of the standard **delay** function.

In order to implement this example with all required modules, utilize the Arduino multi-function shield available in the lab to present the results.

```
void taskBlink1 (void *pvParameters)
{
  while(1)
  {
    MFS.writeLeds (LED_1, ON);
    vTaskDelay (500 / portTICK_PERIOD_MS);
    MFS.writeLeds (LED_1, OFF);
    vTaskDelay (500 / portTICK_PERIOD_MS);
  }
}
```

1 https://www.freertos.org/a00125.html

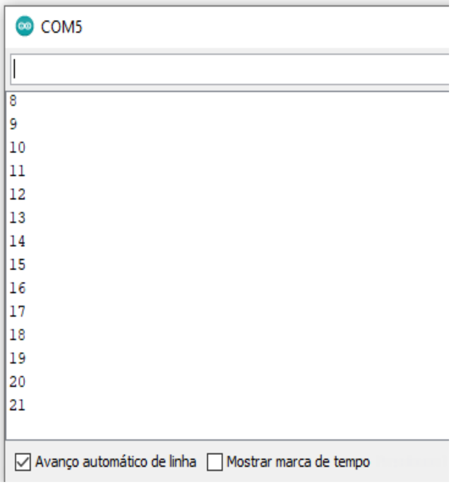
2 https://www.freertos.org/a00132.html The ***vTaskStartScheduler*** must be included in FreeRTOS projects, even though the Arduino IDE is including this routine automatically.

3 https://www.freertos.org/a00127.html

```
void taskPrint (void *pvParameters)
{
  int countTime = 0;
  while(1)
  {
    countTime++;
    Serial.println(countTime);
    vTaskDelay (1000 / portTICK_PERIOD_MS);
  }
}
```

The constant **portTICK_PERIOD_MS** is related to the tick period. The FreeRTOS has a default tick frequency of 1KHz (can be changed be the function configTICK_RATE_HZ). By default, the value of portTICK_PERIOD_MS is “1 ms per tick”.

After properly programmed, test the execution of your multi-tasking code. Open the terminal window and confirm that LED 1 is switching ON/OFF at 1Hz.



```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

Figure 5- Results of “TaskPrint” using the serial monitor.

Exercise 1: Basics of preemption and tasks control

Rewrite the Exercise 1 of Part A as a multi-tasking program (RTOS tasks) without the use of finite state machines.

Exercise 2: Enhanced tasks control

Create the following tasks using the FreeRTOS library:

- ***TaskBlink1*** that toggles LED D1 at 1Hz;
- ***TaskPrint*** that counts seconds and sends the value over UART (9600 Baud Rate) at 1Hz.
- ***TaskDisplay*** that counts the number of clicks on button **S1** (left side of the shield), showing the result in the 7-segment display. The code of this particular task is presented as follows:

```
void taskDisplay (void *pvParameters)
{
  int countClicks = 0;
  while(1)
  {
    byte btn = MFS.getButton();
    if (btn == BUTTON_1_PRESSED)
    {
      countClicks++;
      MFS.write(countClicks);
    }
  }
}
```

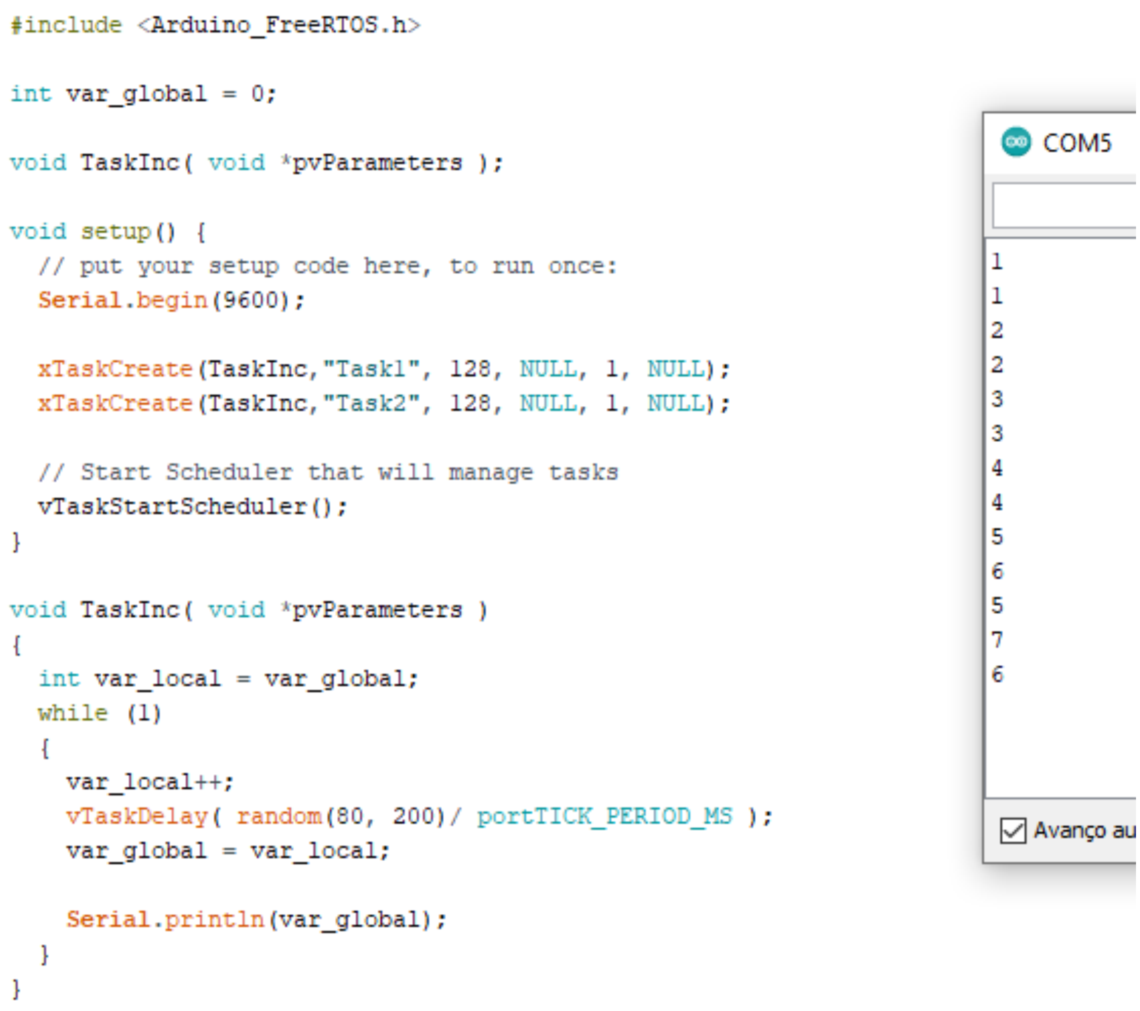
- ***TaskBuzz*** that while **S3** (right side of the shield) is being pressed, suspends ***TaskPrint***, activates the buzzer and prints "*Emergency*" in the serial monitor. ***TaskPrint*** must resume its activity after **S3** is released. The **S3** should be considered as an emergency button.

Part C - Advanced concepts of FreeRTOS

Introduction to Race Conditions and Mutual Exclusion

Race condition is *"a situation in which multiple threads or processes read and write a shared data item, and the final result depends on the relative timing of their execution"* (from Stallings). In practice, race conditions appear when two (or more) tasks need to update a shared global variable but are unable to change its value in a single CPU instruction, thus the global system behavior is dependent on the timing of uncontrollable events.

This can easily happen when global variables are shared among multiple tasks, as described in the following example:



```
#include <Arduino_FreeRTOS.h>

int var_global = 0;

void TaskInc( void *pvParameters );

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);

  xTaskCreate(TaskInc,"Task1", 128, NULL, 1, NULL);
  xTaskCreate(TaskInc,"Task2", 128, NULL, 1, NULL);

  // Start Scheduler that will manage tasks
  vTaskStartScheduler();
}

void TaskInc( void *pvParameters )
{
  int var_local = var_global;
  while (1)
  {
    var_local++;
    vTaskDelay( random(80, 200)/ portTICK_PERIOD_MS );
    var_global = var_local;

    Serial.println(var_global);
  }
}
```

```
1
1
2
2
3
3
4
4
5
6
5
7
6
```

Figure 6 - Example of a race condition.

In this example, the global variable is being accessed by two concurrent threads with different operation cycles (random delays). As a result, that global variable is being accessed in a way that **consistence** is lost, requiring some "treatment".

To prevent race conditions, it is required that when one task is in a **critical section** that accesses shared resources, no other task may be in a critical section that accesses any of those shared resources. In order to solve this issue, Mutex (Mutual Exclusion) is a powerful tool that can be used to control the access of the shared resources, adopting for that a simple concept (see Figure 7). In general, the process that locks the mutex (sets the value to 0) must be the one to unlock it (sets the value to 1).

FreeRTOS provides many functions/structures that make the implementation of a mutex straightforward, namely: ***SemaphoreHandle_t***[4]***, xSemaphoreCreateMutex***[5]***, xSemaphoreTake***[6] *and* ***xSemaphoreGive***[7]***.***

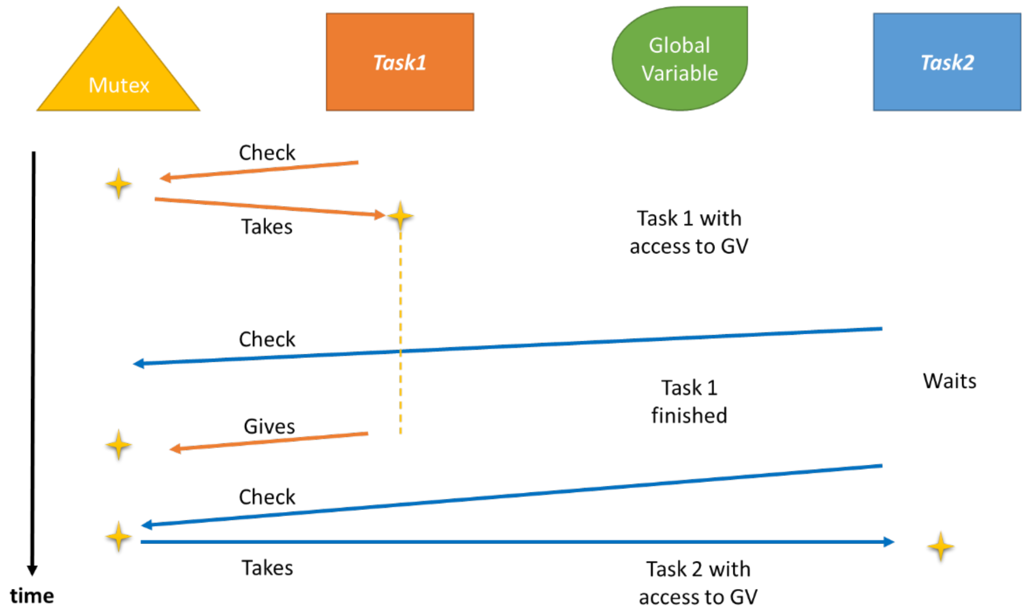


Figure 7 - Mutual exclusion for two tasks accessing the same global variable (shared resource).

Exercise 1 – Implementation of a simple mutex

Implement a mutex in the previous example to guarantee that the variable ***var_global*** cannot be accessed at the same time by multiple tasks. Check the results in the serial port to realize how different the results will be. After that, change the program to display the numerical number on the LCD display of the Arduino multifunction shield.

Exercise 2 – Producer/Consumer tasks with blocking behavior

A common problem in concurrent systems is when one fast system (task) produces something that must be consumed by a different slow task, sequentially. In a simple perspective of this problem, the producer task will produce one token every cycle (that may be random), which is stored in a buffer that support no more than 1 token

[4] https://www.freertos.org/xSemaphoreCreateBinary.html
[5] https://www.freertos.org/CreateMutex.html
[6] https://www.freertos.org/a00122.html
[7] https://www.freertos.org/a00123.html

(shared variable). The consumer task will remove one token from that buffer at each (random) cycle, as long as there is at least one token to be removed. Their basic operation is demonstrated as follows:

```
int sharedBuffer = 0;

//Producer task
void TaskProd (void *pvParameters)
{
  while(1)
  {
    //Take mutex, but do not wait more than 10 ticks
    if (xSemaphoreTake(mutex, 10) == pdTRUE)
    {
      int produzir = sharedBuffer;
      if (produzir == 0)
        produzir++;

      //Simulate some FAST processing to produce something
      vTaskDelay (random(10,100)/ portTICK_PERIOD_MS);
      sharedBuffer = produzir;

      Serial.print(pcTaskGetName(xTaskGetCurrentTaskHandle()));
      Serial.print(": Buffer is ");
      Serial.println(sharedBuffer);

      //Give mutex
      xSemaphoreGive(mutex);
    }
  }
}

//Consumer task
void TaskCons (void *pvParameters)
{
    while(1)
    {
      //Take mutex, but do not wait more than 10 ticks
      if (xSemaphoreTake(mutex, 10) == pdTRUE)
      {
        int consumir = sharedBuffer;
        if (consumir > 0)
          consumir--;

        //Simulate some SLOW processing to consume something
        vTaskDelay (random(500,1000)/ portTICK_PERIOD_MS);

        sharedBuffer = consumir;
        Serial.print(pcTaskGetName(xTaskGetCurrentTaskHandle()));
        Serial.print(": Buffer is ");
        Serial.println(sharedBuffer);

        //Give mutex
        xSemaphoreGive(mutex);
     }
  }
}
```

Execute this program and analyze the output, paying attention to the outcome in the serial port.

Sometimes, very fast tasks may "monopolize" a shared resource, starving other slower tasks of the same priority. A solution for that is to **block** (vTaskSuspend) and **resume (**vTaskResume**)** tasks to allow a fairer access to the resources.

Note: in order to resume a particular task, use the variable of type **TaskHandle_t** as a parameter to the function **vTaskResume**. That variable was passed as the last argument of the function **vCreateTask**.

Exercise 3 – LED Blinking with concurrency

Create the following tasks to be used with the Arduino multifunction shield.

- ***TaskBlink1*** that toggles LED D4 @ variable frequency;
- ***TaskSerial*** that receives a number (1 to 1000) from the serial monitor that defines the switching time in milliseconds of LED D4.
- ***TaskButtonS2*** that multiplies by 2 the switching time of LED D4 when button **S2** is pressed. It shows the switching frequency in a 7-segment display.
- ***TaskButtonS1*** that divides by 2 the switching time of LED D4 when button **S1** is pressed. It shows the switching time in a 7-segment display.
- ***(optional) TaskBuzz*** that increases the intensity of the buzzer from mute to MAX, over a period of time, then it decreases the volume and it loops again.

Exercise 4 – Challenge

Based on Exercise 3, define a high-priority task that handles ALL interactions with the 7-segment display, removing the direct access to the display from the other tasks. That task will be triggered when necessary, doing what it has to do (send commands to the display). Then, it will block itself until a new "call" is received from a normal-priority task (any of the remaining tasks). Remember that since it is a high-priority task, it will execute until its self-blocking command is reached, avoiding all remaining tasks to operate meanwhile.

Annex:

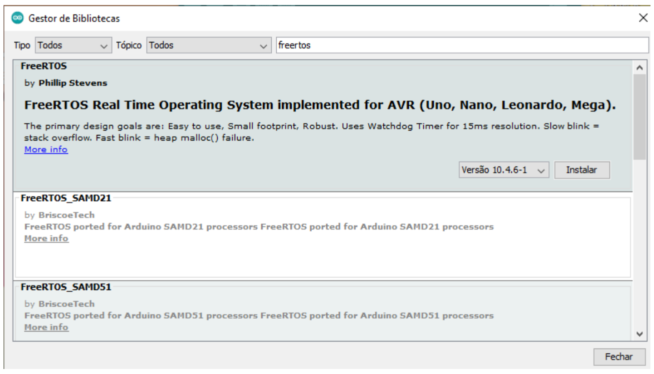
Installing Arduino IDE

Go to download and install the Arduino IDE from the site https://www.arduino.cc/en/software

Installing FreeRTOS Library

Open Arduino IDE and go to ***Sketch -> Include Library -> Manage Libraries***. Search for FreeRTOS and install the library as shown below.

Note: You can also download the library directly from Github and Add the .zip file in ***Sketch-> Include Library -> Add .zip*** file.


Gestor de Bibliotecas
Tipo Todos Tópico Todos freertos

FreeRTOS
by **Phillip Stevens**
FreeRTOS Real Time Operating System implemented for AVR (Uno, Nano, Leonardo, Mega).
The primary design goals are: Easy to use, Small footprint, Robust. Uses Watchdog Timer for 15ms resolution. Slow blink = stack overflow. Fast blink = heap malloc() failure.
More info
Versão 10.4.6-1 Instalar

FreeRTOS_SAMD21
by BriscoeTech
FreeRTOS ported for Arduino SAMD21 processors FreeRTOS ported for Arduino SAMD21 processors
More info

FreeRTOS_SAMD51
by BriscoeTech
FreeRTOS ported for Arduino SAMD51 processors FreeRTOS ported for Arduino SAMD51 processors
More info

Fechar

Installing Arduino MultiFuncShield-Library

Download the library from https://github.com/hpsaturn/MultiFuncShield-Library. Add the .zip file (containing all downloaded files) directly through the Arduino IDE: **Sketch** -> **Include Library** -> **Add .ZIP Library**.