Multiprogramming using an ATMEL Microcontroller

Contents

Introduction

Real-time tasks are critical for embedded devices, where timing plays a very important role. Real-time tasks are *time-deterministic,* meaning that the response time to any event can be guaranteed to be below a value that can be determined at design time. A Real Time Operating System (RTOS) is designed to run applications with tight timing requirements and high reliability. An RTOS also helps in multi-tasking. This laboratory work will be divided into 4 sessions with the following 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****.*

Part D - *Assessment through the development of a small program (individual).*

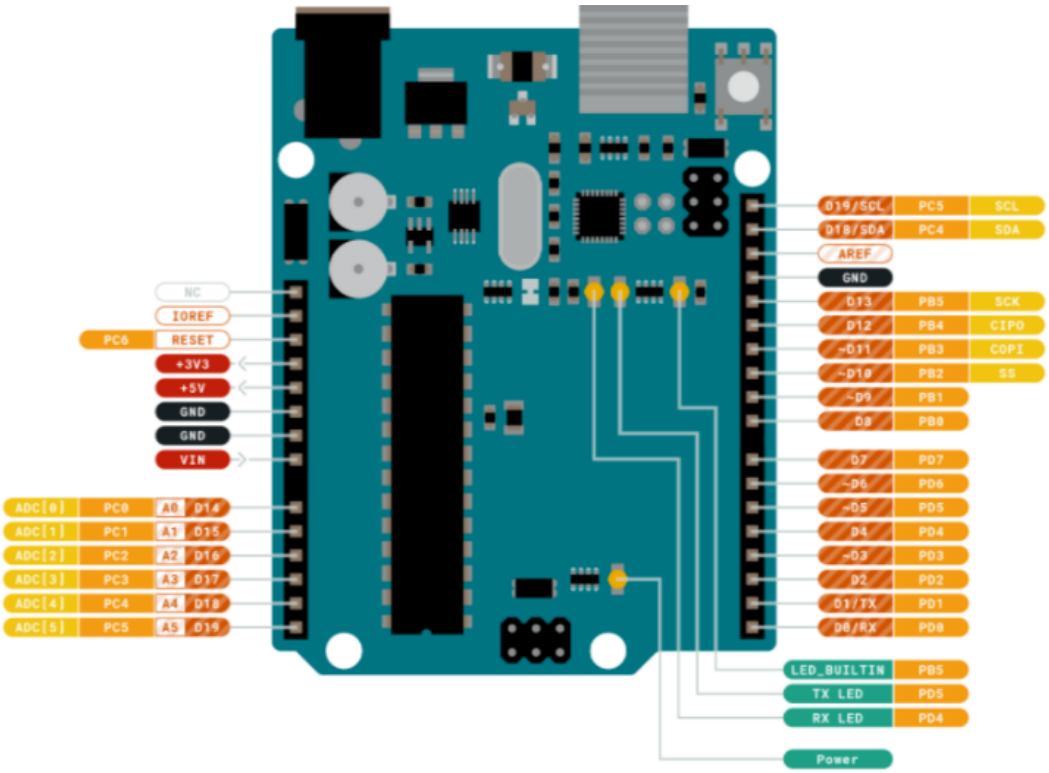
Complementary readings:

- 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.



I/Os	**Built-in LED Pin**	13
	Digital I/O Pins	14
	Analog input pins	6
	PWM pins	6
Communication	**UART**	Yes
	I2C	Yes
	SPI	Yes

Figure 1 - Arduino Uno pinout.

Introduction to the AVR Library:

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.

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. In this way, 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 in the Arduino IDE. Test the examples of ***Program 1*** and ***Program 2*** using the computer that is available in your laboratory (alternatively, you can simply analyze the code).

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
#define LED2 PB2

// Global variables…

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

// Functions needed…

//////////////////////////
// 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) { };
}
```

Concurrency can be implemented with the AVR library by using 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 this example we will use interrupts generated periodically by a Timer unit (Timer 2).

Therefore, an ISR can be used to execute one or more tasks that run concurrently with the main program. This is what happens in the example above. 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".

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 program 2 can be seen in the next figure, but note that the two tasks never interrupt each other.

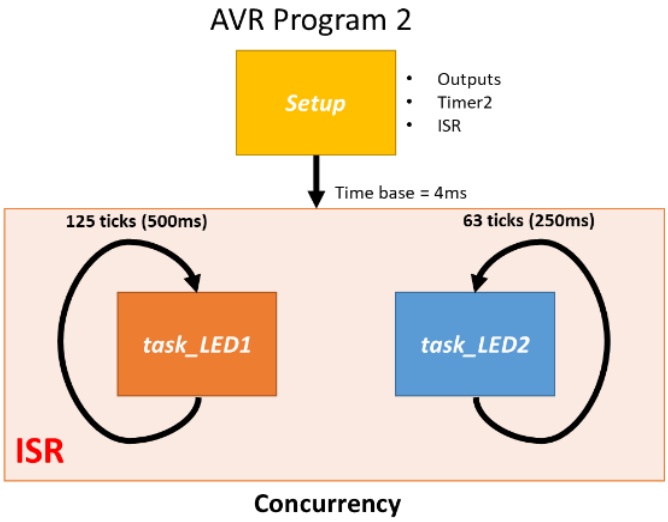


Figure 2 - Graphical representation of program 2. The prescaler for Timer 2 is set to 256, which gives a sample time of 16ms for Timer 2. Since the interruption is generated when Timer 2 reaches the value of 250, the ISR period is 4ms. Then, for task 1 (task 2), the LED state switches after 125 (64)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

Introduction to the Arduino Library:

The Arduino library provides a hardware abstraction that is more user-friendly when compared to the AVR library. 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). Read the slides that support this lecture for understanding how to use this Arduino library.

Arduino Program 1

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

// 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
}
```

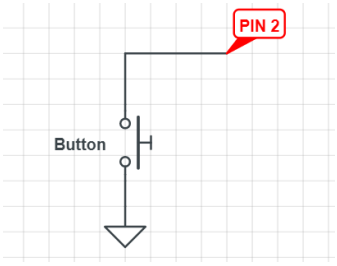
By re-implementing Program 1 using the Arduino library, we can see the enhanced level of abstraction that this library offers, hiding several hardware-related details.

Exercise 1 – Non-blocking LED Switching

The program right above is blinking an LED at 2.5Hz. This is achieved by 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". Now, let us implement a non-blocking program, i.e., without using the "delay" function, and without using interrupt routines. For this purpose, Arduino has a useful time-counting function that abstracts the timer interrupt ticks. The ***millis***[1] function counts time in milliseconds since the last start upon reset. Use it to measure time and implement your own ticks inside the ***loop*** function to control the LED ON/OFF switching. In this way, the CPU will still be available to execute other potential application code (potentially other tasks) inside the ***loop*** function.

Exercise 2 – Interruptions and Digital Inputs

Interrupts[2] are useful for running small routines automatically in microcontroller programs and they can help solve timing problems (e.g., it is a way of executing a routine very quickly after the event that caused the interrupt). Add an external hardware interrupt associated to ***digital pin 2*** to the code of the previous exercise to suspend the switching of the LED. To



[1] https://www.arduino.cc/reference/en/language/functions/time/millis/

[2] https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

generate the interrupt signal, connect a button between the ***digital pin 2*** and the ground reference of the Arduino (use a breadboard if necessary, or a simple wire directly between the Arduino pins) and configure the interrupt to be generated upon a *falling edge* of the input signal. Activate the internal pull-up resistors of the ***digital pin 2*** to stabilize the input signal.

Exercise 3 – Interruptions and Timers

Implement an intermittent alarm (frequency of 1Hz) using***Timer1*** of Arduino (along with the ***TIMER1_OVF_vect*** interruption).The alarm must be suspended or activated when the button ***S1*** (leftmost) of the *MultiFunctionShield* is pressed. The following code aims at helping you configuring the ***Timer1***.

```
const int ledPin        =  3;
const int ledBut        =  15;

bool buzz_isactive      = true;

void setup()
{
  pinMode(ledBut, INPUT);
  pinMode(ledPin, OUTPUT);

  // initialize timer1
  noInterrupts();           // disable all interrupts
  TCCR1A = 0;
  TCCR1B = 0;

  TCNT1 = 34286;            // preload timer 65536-16MHz/256/2Hz
  TCCR1B |= (1 << CS12);    // 256 prescaler
  TIMSK1 |= (1 << TOIE1);   // enable timer overflow interrupt
  interrupts();             // enable all interrupts
}
```

Complete the ***loop*** and the ***ISR(TIMER1_OVF_vect)*** functions accordingly. Use the ***ISR*** to toggle ON and OFF the ***Buzzer*** in *MultiFunctionShield.* Use the pattern proposed in the previous exercise to add a task (invoked within ***loop***) to read the state of the S1 button. Try with different task periods (defined when invoking it inside ***loop***) and see the resulting behavior, particularly when ***S1*** is pressed rapidly.

Note that this exercise shows how it is sometimes convenient to include low abstraction code together with higher abstraction code, for the sake of controlling specific hardware features (the timer configuration in this case).

Exercise 4 – A scheduler of multiple tasks inside the loop() function

Building on the previous exercise, let us do the same structure as AVR Program 2 now using the Arduino library. Take the code inside the ***loop*** function in the previous exercise and move it to a function called ***Task1_blink***. Then, inside ***loop*** you just

call ***Task1_blink***. Now add another function called ***Task2_time*** that writes to the serial port the current time returned by ***millis*** with a period of 1s. Inside ***loop*** now invoke both ***Task1_blink*** and ***Task2_time***. This is a very simple multitasking structure similar to AVR Program 2, but with a higher abstraction level.

Consider using the following pattern as a general way of executing a certain task based on absolute time (in this case given by ***millis***). Note that the actual period, in ms, is passed as a function parameter when the task is called.

```
void Task2_time(long period) {
  static long now =0; // variable to control time

  // activation pattern
  if (millis() >= now + period){
    now = millis();

    // here goes the code of the task
    Serial.println(now);
  }
```

Part B - Concepts of FreeRTOS

Introduction to the FreeRTOS Library:

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

- ***TaskBlink1*** that toggles LED D1 @1Hz;
- ***TaskPrint*** that counts seconds and sends the value over UART (9600 Baud Rate) @ 1Hz.

Both the two tasks will run concurrently (and permanently). The FreeRTOS makes it possible to avoid a sequential execution of routines which is crucial for running multiple tasks at the same time. It is physically possible to run multiple tasks at the same time in a multicore processor. However, this microcontroller is single-core, thus the CPU time needs to be split onto different tasks ("*time multiplexing*")

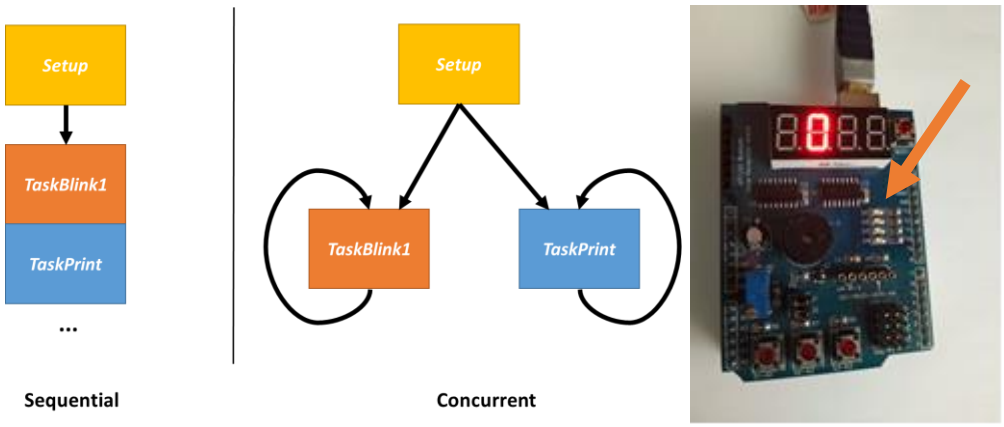


Figure 3 - Execution of two routines (serial communication and LED Blink) using a sequential or concurrent execution.

Import the Arduino FreeRTOS header and create the function prototype for each task that will be executed.

```
#include <Arduino_FreeRTOS.h>

void TaskBlink1( void *pvParameters );
void TaskPrint( void *pvParameters );
```

Initialize the serial communication (9600 Baud Rate), and create two tasks, using the FreeRTOS API, in the ***setup()*** function. Explore the code below, in particular, ***xTaskCreate***[3] and ***vTaskStartScheduler***[4]***.***

[3] https://www.freertos.org/a00125.html

[4] https://www.freertos.org/a00132.html The ***vTaskStartScheduler*** must be included in FreeRTOS projects however, the Arduino IDE is including this routine automatically.

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

  // Create Task for toogle D1
  xTaskCreate(
              TaskBlink1,   // Function to be called
              "Task1",      // Name of task for debuging
              128,          // Stack size
              NULL,         // Parameter to pass to function
              1,            // Task priority (0 to configMAX_PRIORITIES -1)
              NULL);        // Task handle

  // Create Task for Serial Coms
  xTaskCreate(TaskPrint,"Task3", 128, NULL, 1, NULL);

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

Implement functions ***TaskBlink1*** and ***TaskPrint*** according to the Arduino Uno and Shield configuration, as shown below. Explore the code, in particular, the ***vTaskDelay*[5]*.

```
void TaskBlink1(void *pvParameters)
{
  pinMode(LED_BUILTIN, OUTPUT);
  while(1)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay( 500 / portTICK_PERIOD_MS );
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay( 500 / portTICK_PERIOD_MS );
  }
}

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

[5] https://www.freertos.org/a00127.html

Compile and upload the program to Arduino. Open the terminal window and confirm that LED D1 is switching ON/OFF at 1Hz.

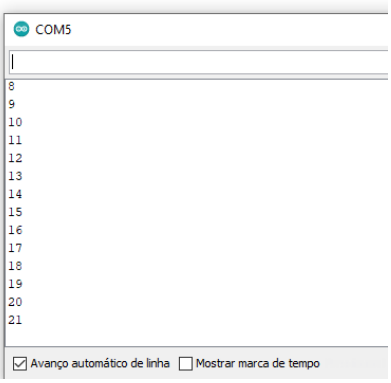


Figure 4- Results of "TaskPrint" using the serial monitor.

Exercise 1: Preemption and Task control

Create the following tasks using the FreeRTOS and Arduino IDE:

- ***TaskBlink1*** that toggles LED D1 @1Hz;
- ***TaskPrint*** that counts seconds and sends the value over UART (9600 Baud Rate) @ 1Hz.
- ***TaskDisplay*** that counts the accumulated number of seconds on which button **S1** (left side of the shield) is pressed and shows the result in the 7-segment display. The code of this task is presented below:

```
void TaskDisplay(void *pvParameters)
{
  MultiFunctionShield MFS;
  MFS.begin();
  pinMode (BUTTON_1_PIN, INPUT);    // insert J2 on board for pull-ups!

  long counter = 0;
  while(1)
  {
    if (!digitalRead(BUTTON_1_PIN)){
        counter++;
        MFS.Display(counter);
    }
    vTaskDelay (1000/ portTICK_PERIOD_MS);
  }
}
```

- ***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** being released. The **S3** should be

considered as an emergency button.
Note: explore the functions ***vTaskSuspend***[6] and ***vTaskResume***[7].

[6] https://www.freertos.org/a00130.html
[7] https://www.freertos.org/a00131.html

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, see the code below.

```
#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);
  }
}
```

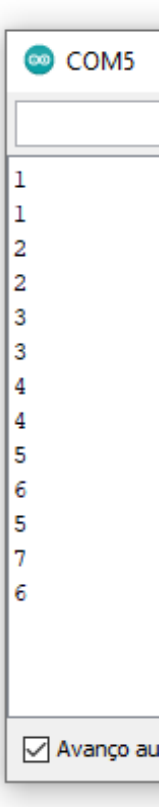


Figure 5 - Example of a race condition.

To prevent race conditions, it is required that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources. Mutex (Mutual Exclusion) is a powerful but

simple concept (see figure below). 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***[8]***, xSemaphoreCreateMutex***[9]***, xSemaphoreTake***[10] *and* ***xSemaphoreGive***[11]***.***

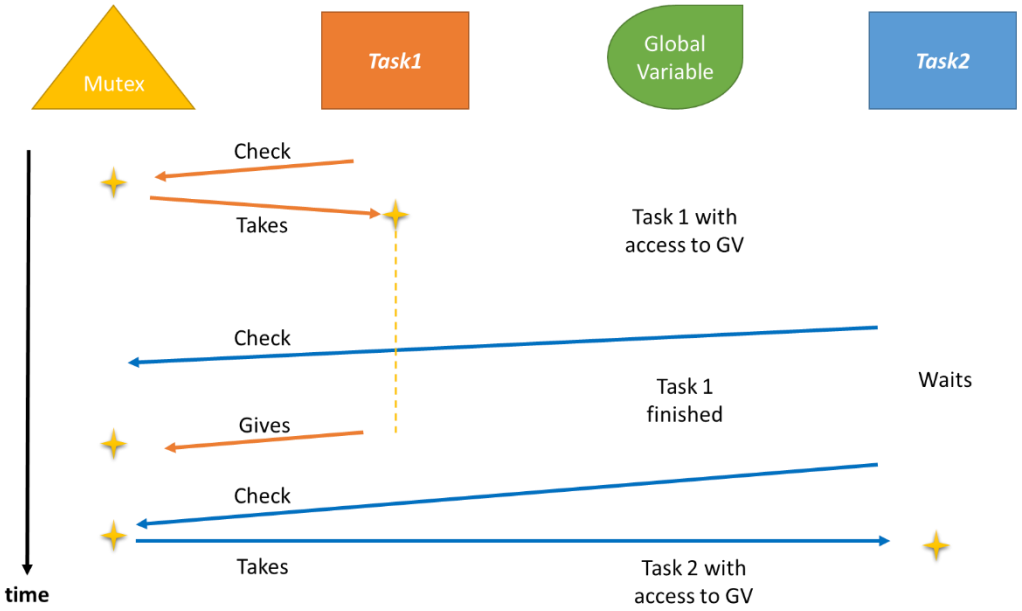


Figure 6 - Mutual exclusion for two tasks accessing the same global variable (shared resource).

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.

Exercise 2 – LED Blinking

Create the following tasks using the FreeRTOS and Arduino IDE:

- ***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.

[8] https://www.freertos.org/xSemaphoreCreateBinary.html
[9] https://www.freertos.org/CreateMutex.html
[10] https://www.freertos.org/a00122.html
[11] https://www.freertos.org/a00123.html

- ***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..

Annex:

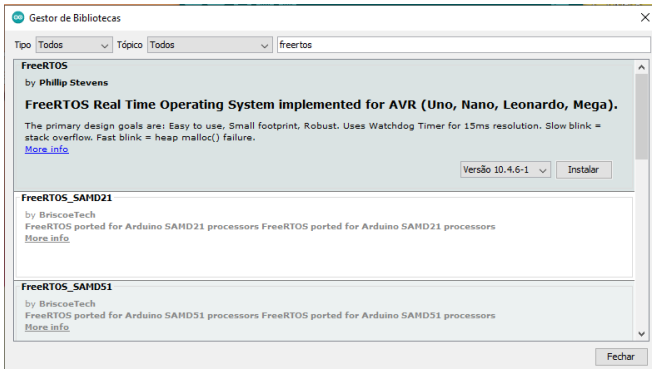
Installing Arduino IDE

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

Installing FreeRTOS Library

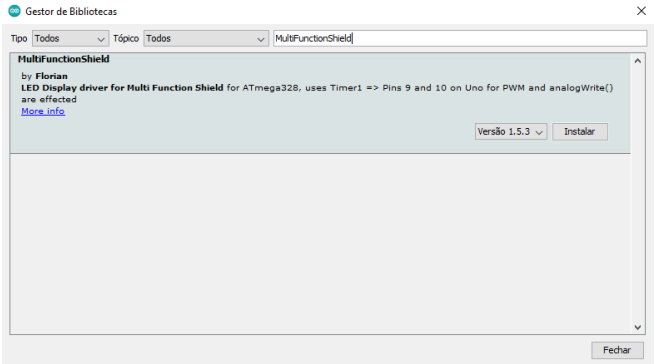
Open Arduino IDE and go to ***Sketch -> Include Library -> Manage Libraries***. Search for FreeRTOS and install the library as shown below.

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



Installing Arduino MultiFunctionShield Library

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



Restart the Arduino IDE.