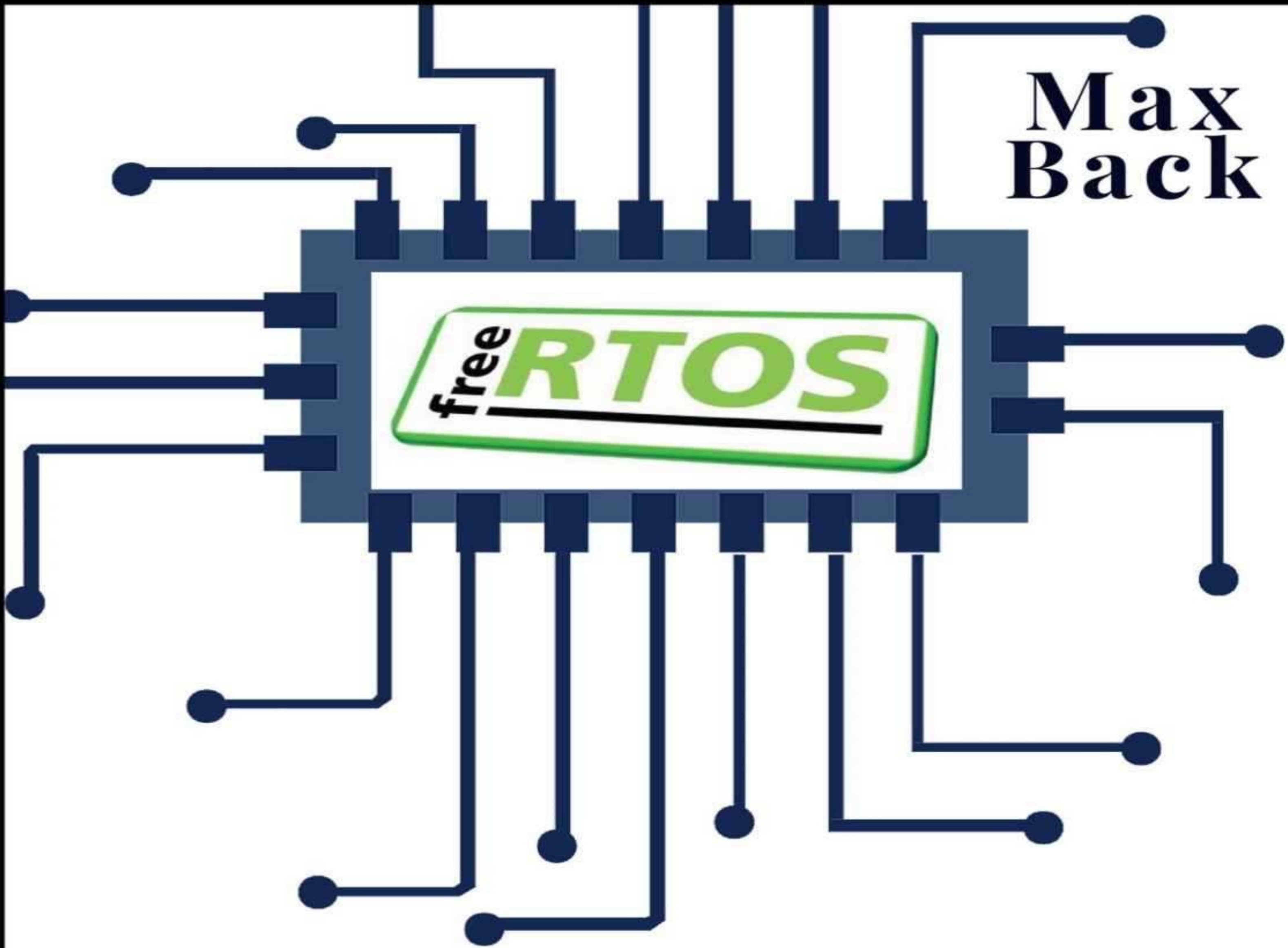
Max Back



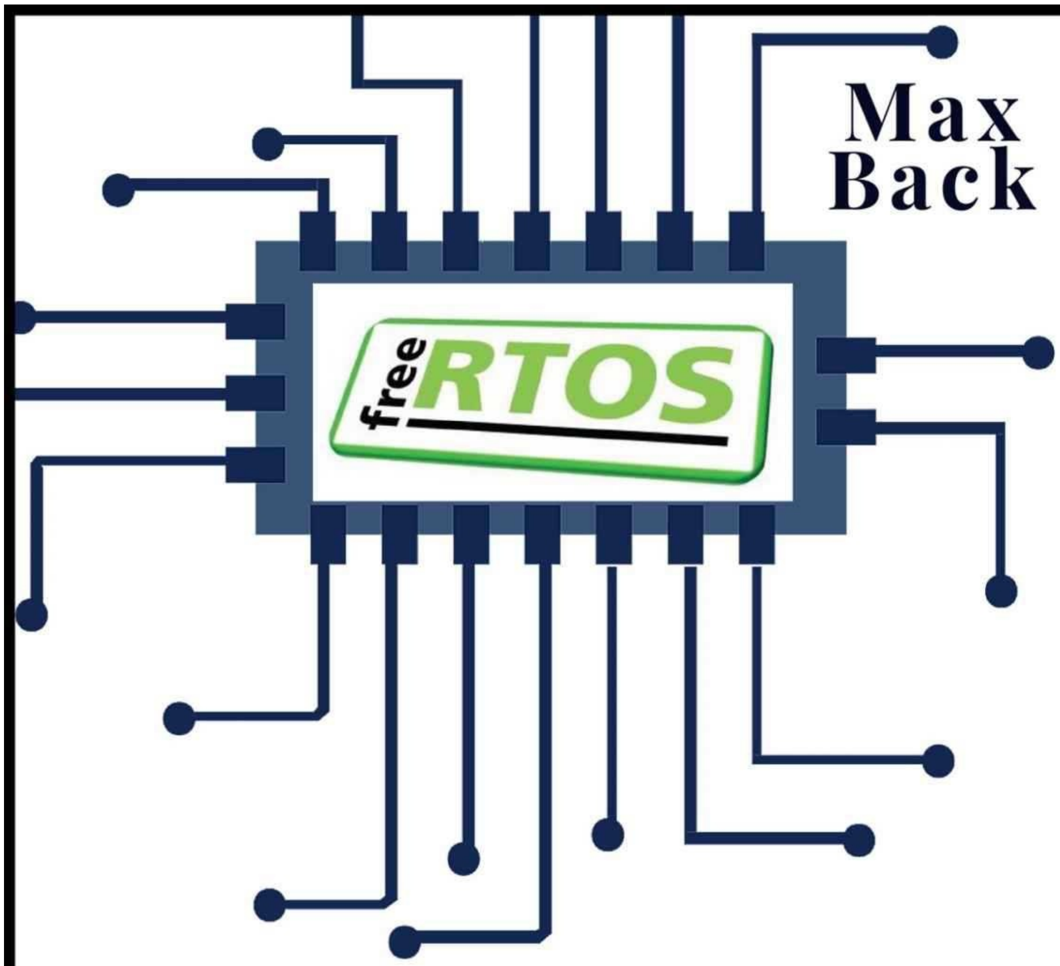
FREERTOS

A PRACTICAL APPROACH WITH ARDUINO

(Programando Multitarefa na prática - Utilizando a linguagem C/C++, freeRTOS e Arduino)

All examples are written for Arduino and their code is explained in detail.

Max
Back

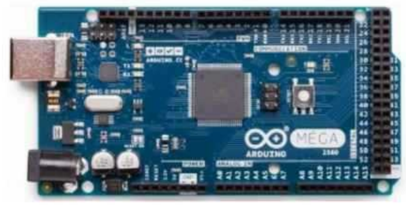


FREERTOS

A PRACTICAL APPROACH WITH ARDUINO

(Programando Multitarefa na prática - Utilizando a linguagem C/C++, freeRTOS e Arduino)

All examples are written for Arduino and their code is explained in detail.



freeRTOS
A practical approach with Arduino

Max Back

Sumário

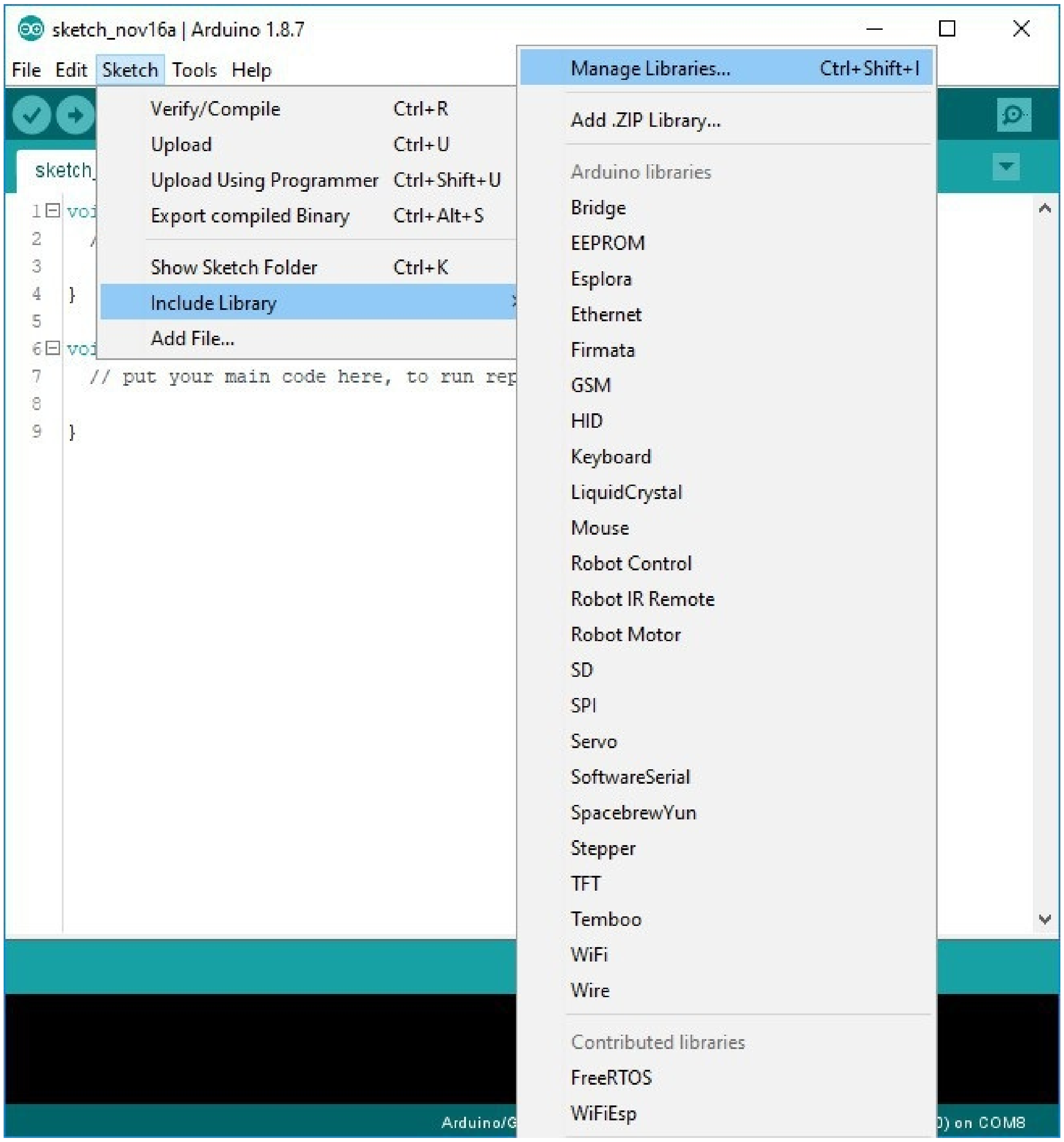
Chapter 1 - Installing freeRTOS for Arduino

Note: You must already have the Arduino IDE installed. If this is not the case, you can obtain it at the following address:
https://www.arduino.cc/en/Main/Software
Choose the most appropriate IDE version and start the download. Just install it and follow the freeRTOS installation instructions, shown below.

Let's start by installing the freeRTOS library from the Arduino. You must be connected to the internet, since the library will be downloaded during the installation process.

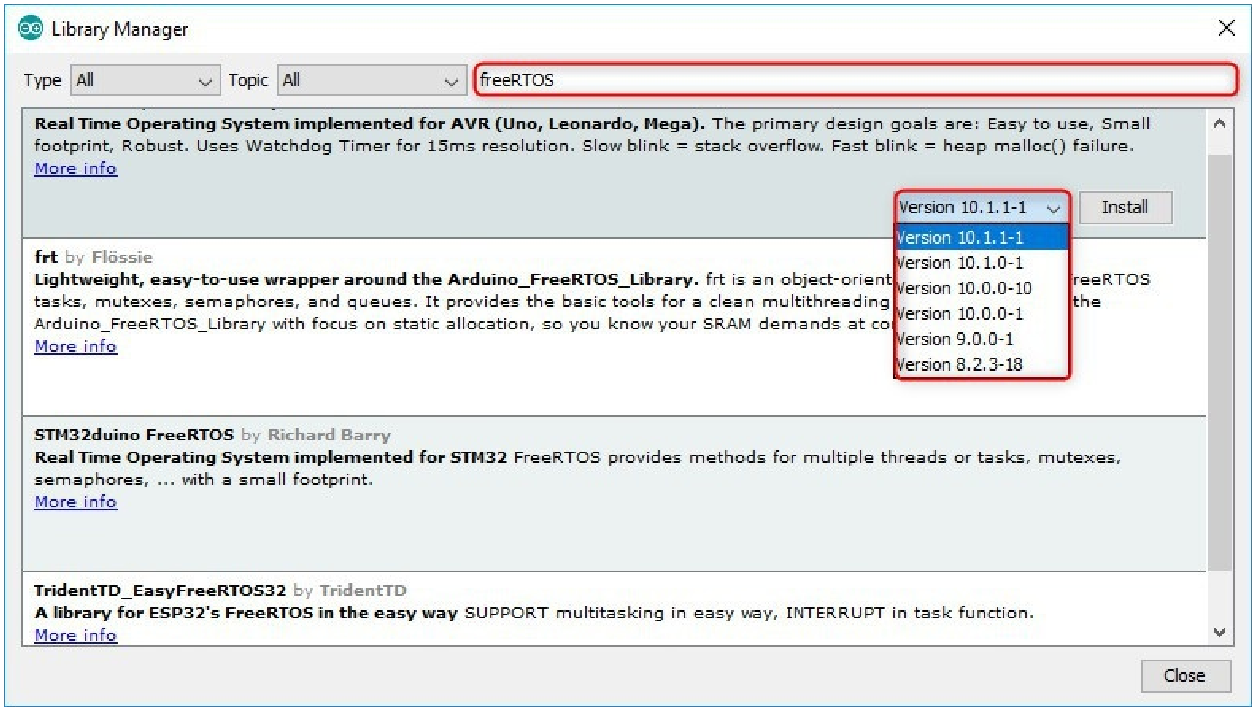
Note: Read all the steps carefully before you start and then use them as a guide if you prefer.

Step 1: Open the Arduino and open then menu item: **Sketch \ Include Library \ Manage Libraries ...** (as shown below):



```
voi
  /

}

voi
  // put your main code here, to run rep

}
```

Step 2: In the text box to refine the search, type freeRTOS (as highlighted in the image) and select the item for AVR (Arduino Uno, Leonardo, Mega). Then select the last version (checkbox next to the install button).


Library Manager

Type All Topic All freeRTOS

Real Time Operating System implemented for AVR (Uno, 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

Version 10.1.1-1 Install

Version 10.1.1-1
Version 10.1.0-1
Version 10.0.0-10
Version 10.0.0-1
Version 9.0.0-1
Version 8.2.3-18

frt by Flössie
Lightweight, easy-to-use wrapper around the Arduino_FreeRTOS_Library. frt is an object-orient reeRTOS tasks, mutexes, semaphores, and queues. It provides the basic tools for a clean multithreading the Arduino_FreeRTOS_Library with focus on static allocation, so you know your SRAM demands at co
More info

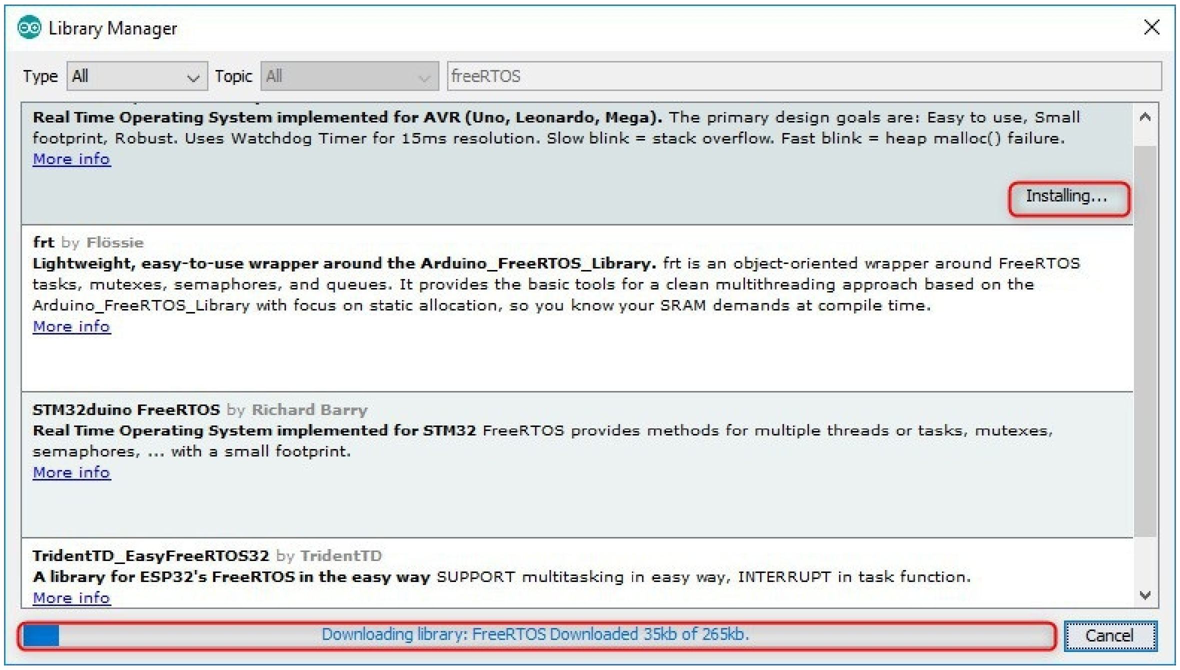
STM32duino FreeRTOS by Richard Barry
Real Time Operating System implemented for STM32 FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores, ... with a small footprint.
More info

TridentTD_EasyFreeRTOS32 by TridentTD
A library for ESP32's FreeRTOS in the easy way SUPPORT multitasking in easy way, INTERRUPT in task function.
More info

Close

Attention: All of the examples in the book were written with version 10.0.0-8 of this library and retested with 10.1.1.-1 version.

Step 3: Press the "Install" button and wait while the library is downloaded.



Real Time Operating System implemented for AVR (Uno, 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

Installing...

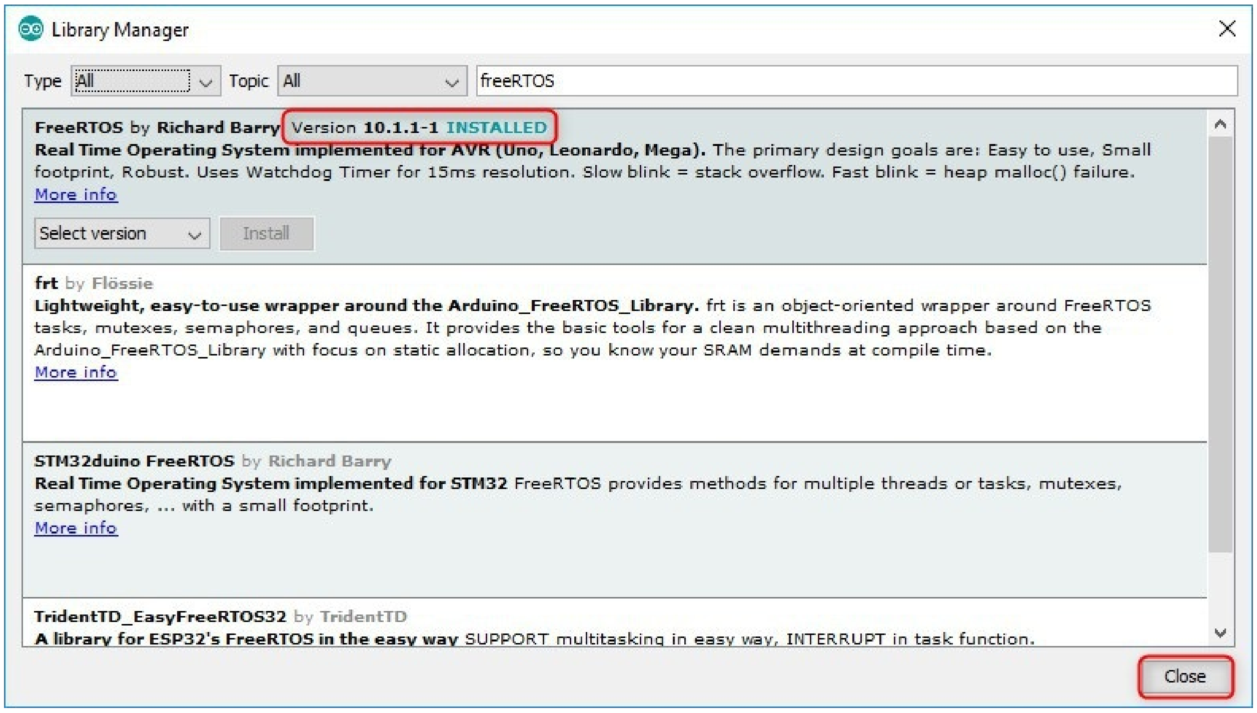
frt by Flössie
Lightweight, easy-to-use wrapper around the Arduino_FreeRTOS_Library. frt is an object-oriented wrapper around FreeRTOS tasks, mutexes, semaphores, and queues. It provides the basic tools for a clean multithreading approach based on the Arduino_FreeRTOS_Library with focus on static allocation, so you know your SRAM demands at compile time.
More info

STM32duino FreeRTOS by Richard Barry
Real Time Operating System implemented for STM32 FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores, ... with a small footprint.
More info

TridentTD_EasyFreeRTOS32 by TridentTD
A library for ESP32's FreeRTOS in the easy way SUPPORT multitasking in easy way, INTERRUPT in task function.
More info

Downloading library: FreeRTOS Downloaded 35kb of 265kb.

Step 4: If all goes well, the installed version information will be displayed. Just press the close button.



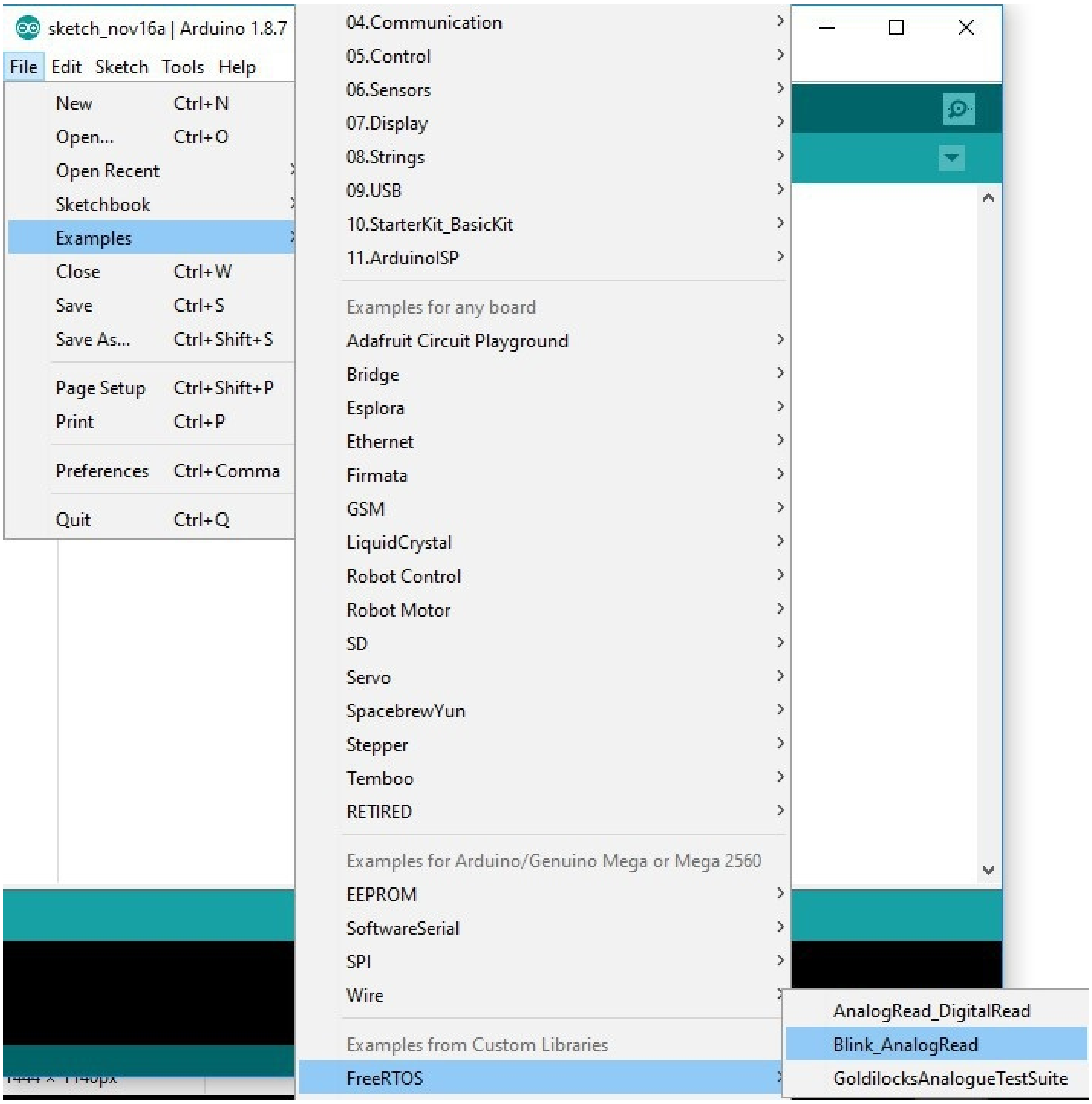
FreeRTOS by **Richard Barry** Version **10.1.1-1** **INSTALLED**
Real Time Operating System implemented for AVR (Uno, Leonardo, Mega). The primary design goals are: Easy to use, Small footprint, Robust. Uses Watchdog Timer for 15ms resolution. Slow blink = stack overflow. Fast blink = heap malloc() failure.
More info

frt by Flössie
Lightweight, easy-to-use wrapper around the Arduino_FreeRTOS_Library. frt is an object-oriented wrapper around FreeRTOS tasks, mutexes, semaphores, and queues. It provides the basic tools for a clean multithreading approach based on the Arduino_FreeRTOS_Library with focus on static allocation, so you know your SRAM demands at compile time.
More info

STM32duino FreeRTOS by Richard Barry
Real Time Operating System implemented for STM32 FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores, ... with a small footprint.
More info

TridentTD_EasyFreeRTOS32 by TridentTD
A library for ESP32's FreeRTOS in the easy way SUPPORT multitasking in easy way, INTERRUPT in task function.

Step 5: In the arduino IDE we will then load a sample project, which comes with the freeRTOS library. So we have already verified that it is present in the IDE: Access the menu File \ Examples \ FreeRTOS \ Blink_AnalogRead:



Ready!

A project should open, with the Arduino lib include and two tasks, one to blink the LED of the Arduino board and another to send readings of one of the analog inputs through the serial.

Primeiras linhas do arquivo Blink_AnalogRead.ino:

```
#include <Arduino_FreeRTOS.h>

// define two tasks for Blink & AnalogRead
void TaskBlink( void *pvParameters );
void TaskAnalogRead( void *pvParameters );
```

These are just the first few lines of the source file that was opened. We'll get back to it soon (you can leave it open if you want to), but first we need to cover a few minutes for some basics in the next chapter, which may be useful for better understanding some of the terms that will be used throughout this book.

In the next chapters we will work from standard Arduino examples and the installed library, then creating new projects from scratch.

Chapter 2 - Real-Time Operating System Basics

Multitasking Operating Systems

There are monotask and multitasking operating systems. An example of a single-function operating system is Microsoft's old MS-DOS. There was an operating system, since your program did not run directly on the hardware, as we do firmware (or baremetal). In fact there is an operating system playing the important intermediary role between the complex hardware of a computer and our program.

However, unlike windows, linux and even the operating system of our smartphones, it allowed only one user application at a time. We can even write some more software by creating a driver or device drivers, in order to treat some low level function (such as a card installed on your PC or a serial port), including using operating system interrupts or hardware, in parallel with program you are running.

Multitasking systems allow more than one program, or task, to be started at a time. If there is only one processor and there is only one core in this processor, these programs will not run simultaneously, but will alternate CPU time, which runs one program, and another time. For the user the illusion is that everything is being carried out in parallel, performing tasks such as downloading an internet program and saving to disk, playing a song from the HD and receiving the keyboard and mouse inputs in a text editor, while you receive a notification that you have just received an instant message from one of your contacts.

If there are multiple cores and processors the programs can run in parallel, but not all at the same time, obviously.

Task

Tasks as they are handled in freeRTOS, are the equivalent of your computer programs. Let's think this in a way directed to programming in C, for console: The typical program of C / C ++ works in colleges is the one that has the function main () and inside has an infinite loop, which asks for options for the user, responding to each option with the execution of a specific code snippet and then returning to a new loop interaction, which again asks for a new option.

This example is monotonous, because although it has several options to do, only one is done at a time, in a single line and sequence of instructions. We can also think of a program for a microcontroller with a main () function, where the pins are configured as inputs and outputs, just like a serial is configured and activated, then stays in the loop repeating a code to flash the loop and then send the status of an entry for an entry.

Note: You will notice that the program excerpt uses the Arduino functions, both to configure the pins and other functions. If we stop to repair, this program separates the configuration part at the beginning of the main () function and execution body that is received inside the for (;;) loop.

Whoever uses Arduino does not have access to the main () function. It always implements the setup () and loop () functions, which in practice is to have a way of having access to the boot execution and from within the main loop of the Arduino program without having to see the low-level details of the platform.

Listing 1 - Example of a C monotask application for microcontroller

```
void main(void) {

   //Sets pin 13 as output (LED) and pin 1 as input
  pinMode(13, OUTPUT);
  pinMode(1, INPUT);
06:
07:    //Starts the serial with 9600 bps
08:   Serial.begin(9600);
09:
10:    //main loop
11:   for(;;) {
12:     digitalWrite(13, HIGH);
13:     delay(1000);
14:     digitalWrite(13, LOW);
15:     delay(1000);
16:
17:      //Sends the state of the input key on pin 1
18:     if(digitalRead(1) == HIGH) {
19:       Serial.println("Key on!");
20:     } else {
21:       Serial.println("Key off!");
22:     }
16:   }
17: }
```

The point to note is that this program does two things (after having everything set up):

- Blink the LED connected to the port 13.
- Sends the button status (on or off) via the serial.

These two tasks to be performed, hour one, time another, are performed even if it is a single program. To solve the problem we are looking for some kind of solution. This solution presented is a good starting point, in the sense that it seeks to find a way to do what it takes .

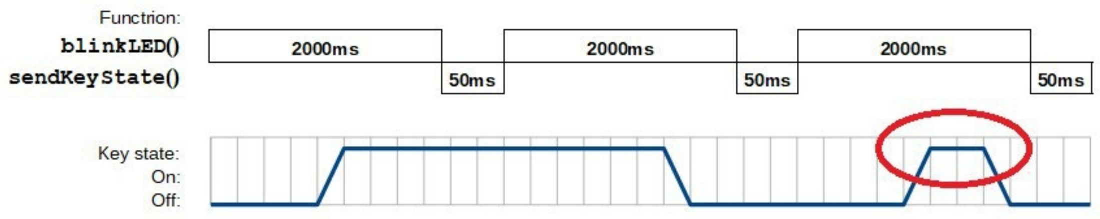
We could say that the code between lines 12 and 15 is one of the tasks, while the code between lines 18 and 22 corresponds to another task.

If we created two functions, each one to encapsulate the code of one of these tasks, we would have a more organized code. Here is the code redone, showing the two functions in bold, as well as your calls:

Listing 1b - Modified monotask application

```
01: void blinkLED(void) {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}

void sendKeyState(void) {
  if(digitalRead(1) == HIGH) {
    Serial.println("Key on!");
  } else {
    Serial.println("Key off!");
  }
}

void main(void) {

  //Sets pin 13 as output (LED) and pin 1 as input
  pinMode(13, OUTPUT);
  pinMode(1, INPUT);

  //Starts the serial with 9600 bps
  Serial.begin(9600);

  //main loop
  for(;;) {
     blinkLED();
    sendKeyState();
  }
}
```

In Listing 1b we have a better organization of the code, but we continue with the execution characteristics. As with the program, a change in the key followed by a reversal of this change (turn on and off afterwards) would not be noticed if it happened while the LED is blinking, ie while in the **`blinkLED`** () function.



In this example we assume that the `blinkLED()` function will take 2000ms (two seconds) and the function `sendKeyState()` 50ms.

During the third call to the `blinkLED()` function the key is turned on and off, causing the event to be ignored, so that the text sent by the would be the following:

```
Key on!
     Key off!
     Key off!
```

However, what really should have been sent to the user would be as follows (the difference is underlined):

```
Key on!
     Key off!
     Key on!
     Key off!
```

The solution to keep a detectable event, even if performing another task other than a direct reading of its value, is to use a hardware interrupt, in which one could change the value of a global variable that would then be read in the function. Interrupts are task types, which allow you to write code (as short as possible) to be executed when an event is detected, such as a change in the state of an input pin. In a way it's a step in the direction of multitasking programming.

The use of interrupts is not eliminated in programming with an RTOS, but is made easier because we can use more robust mechanisms to "talk" between interrupts and written tasks to handle these data and events. Thus manual and potentially dangerous mechanisms, as is the case of a global variable, which has to be tested in the main loop.

Following is a pseudocode that shows the change handling function of the key and another one of a task that treats this value. The hardware configuration details and the interrupt routine are omitted:

Listing 2 - Pseudocode of a task in conjunction with an interrupt

```
01: void handleInputPinInterrupt(void) {
02:    //Sends key status information
03:   sendInformationToTask(digitalRead(1));
04: }
05:
06: void taskSendKeyState(void) {
07:
08    //Local variable to read information
09   char keyState ;
10:
11:   //main task loop
12:   for(;;) {
13:      //Expects to get some information for the task
14:     keyState = receiveInformationFromTask();
15:
16:     if( keyState == HIGH) {
17:       Serial.println("Key on!");
18:     } else {
19:       Serial.println("Key off!");
20:     }
21:   }
22: }
```

We have between the lines 1 to 3 the definition of an interrupt handling function, which will be called each time the pin connected to our switch changes. This routine just reads the new key value and calls a (fictitious) multitasking support function to send this value to the task named `taskSendKeyState()` , which starts at line 6.

In this way, the interruption lasts the shortest possible time, returning to the previous code point, for example. When the time comes to execute the task, it receives the information with the function `receiveInformationFromTask()` , also fictitious. Then decide what text to send by serial.

The task's function looks like a main () function, with an infinite loop at its core, evidencing that it executes without having to bother cooperating with other functions and tasks that may exist. It will be the operating system itself that will momentarily stop the execution of the task code to give chance to another task. This can either occur from time to time or when the current task waits for something that has not yet arrived, for a time or indefinitely.

At line 14, the call to receive an information is a call configured to wait indefinitely. So this task gets frozen until something is sent to it, which in our case is when the state of the key changes and the interrupt handles the change.

The programming is much more localized in this way, because the task has a well defined interface with the outside world: interrupts, I / O pins, other tasks, etc. Looking at this task we can see that it waits for each information, doing nothing until something arrives. Then it sends the appropriate text to the serial and in the next interaction the loop returns to wait again, and so on.

In other more complex cases, one can use RTOS functions with the option of waiting for some time for information, writing an appropriate treatment in the event that nothing happens, ie no data reaches this range.

We can also have tasks that just perform something, unrelated to the rest of the program, like the blinking LED action in our previous example. In this case the task can be executed from the beginning only to indicate that the board is running some software. We could change this task to receive changes of the delay value in order to change the flashing frequency of the LEDs, indicating error, for example.

In fact we can divide various tasks of our systems into tasks and organize the exchange of data with operating system mechanisms such as traffic lights and queues so that the execution of each part of the program is given at the right time.

There is still the issue of task priorities, so that higher priority tasks are more likely to perform than lower priority tasks. Thus, each task change that is performed analyzes the task to be performed next. If a task of higher priority than the others is ready to execute (because the expected information has arrived, for example) it will be executed before the others. There are even mechanisms to make the task that was running when an interrupt happened is not necessarily the task that will continue to run after the interruption.

Imagine that our change is just one of several tasks and that at the time of the key interruption another task is running (and would continue for a few fractions of a second). Well, if the task that treats the key has low priority, it might not be executed next, but if its priority is the highest, it will certainly run immediately after the interrupt. So we can define tasks in which the idea of real time is important and others in which it is not so much so. This is, in broad terms, the idea behind the terms *hard real time* e *soft real time* .

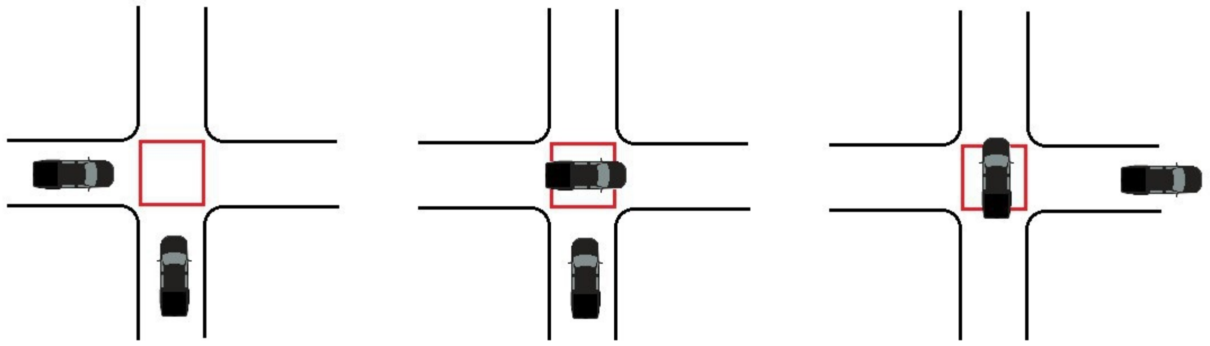
I tried to approach some of the concepts that might be better understood in practice throughout the book and in reading additional material, such as the freeRTOS manuals, whose link was added at the beginning of this book.

Semaphore

Traffic lights are signaling systems that allow you to coordinate the execution between more than one task. It is through them that the tasks can perform signaling, without necessarily having any associated data, just the context associated by the programmer at the semaphore itself.

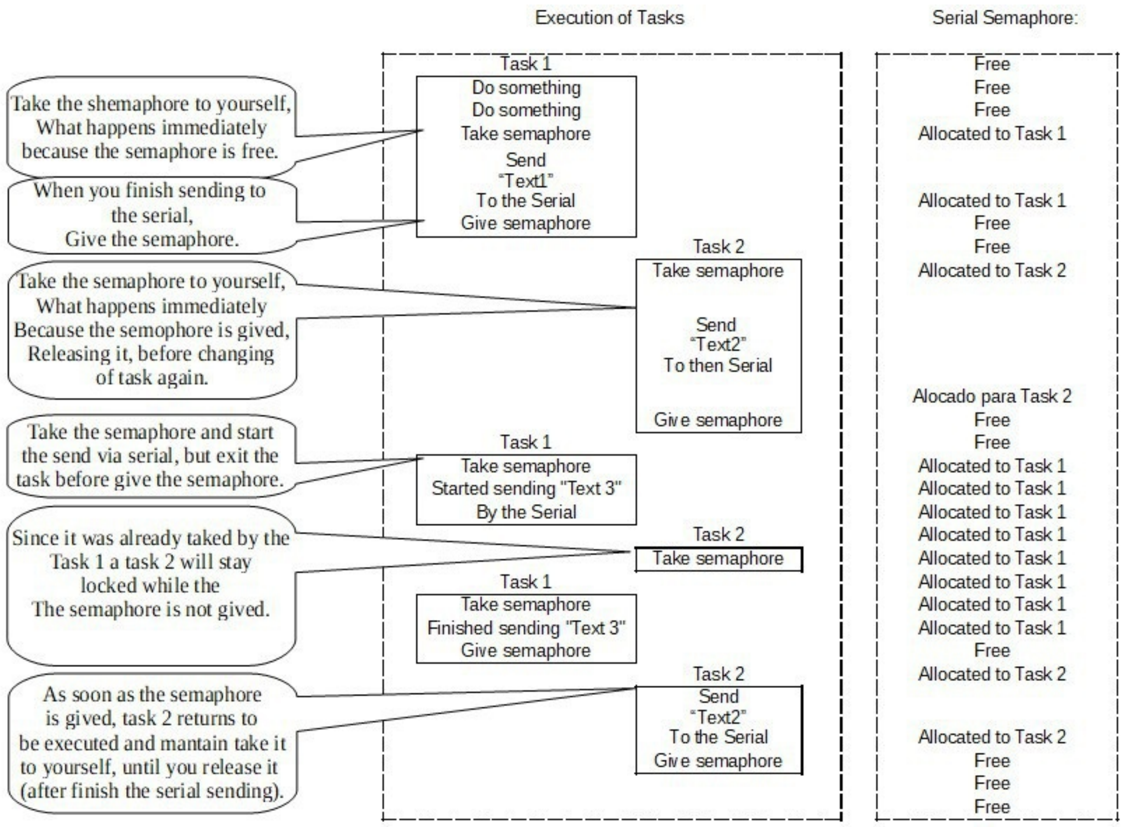
One of the types of traffic lights are mutually exclusive, we call MUTEX (MUTual EXclusion), that is, it signals as a context in which only one of the tasks can act is occurring, and that all others that pretend to act in that context are excluded, like a crossing of streets with a semaphore.

In this example, the fact that one sense passes through the intersection excludes the possibility of the other sense also passing through. In a second instant the second direction will use the intersection, which excludes the possibility that the cars in the first direction cross and collide.



This object is known by all tasks interested in using this context, and has the concurrency controls when accessing the object guaranteed by the operating system implementation, which guarantees (by itself the one that changes the state of the traffic light through its call) that two tasks will not catch the context for them simultaneously, causing collisions.

The following image illustrates the use of a MUTEX traffic light to control the access of more than one task to the sending functions by the arduino serial.


Execution of Tasks
Serial Semaphore:
Take the shemaphore to yourself, What happens immediately because the semaphore is free.
When you finish sending to the serial, Give the semaphore.
Take the semaphore to yourself, What happens immediately Because the semophore is gived, Releasing it, before changing of task again.
Take the semaphore and start the send via serial, but exit the task before give the semaphore.
Since it was already taked by the Task 1 a task 2 will stay locked while the The semaphore is not gived.
As soon as the semaphore is gived, task 2 returns to be executed and mantain take it to yourself, until you release it (after finish the serial sending).
Task 1
Do something
Do something
Take semaphore
Send "Text1" To the Serial
Give semaphore
Task 2
Take semaphore
Send "Text2" To then Serial
Give semaphore
Task 1
Take semaphore
Started sending "Text 3" By the Serial
Task 2
Take semaphore
Task 1
Take semaphore
Finished sending "Text 3"
Give semaphore
Task 2
Send "Text2" To the Serial
Give semaphore
Free
Free
Free
Allocated to Task 1
Allocated to Task 1
Free
Free
Allocated to Task 2
Alocado para Task 2
Free
Free
Allocated to Task 1
Allocated to Task 1
Allocated to Task 1
Allocated to Task 1
Allocated to Task 1
Allocated to Task 1
Allocated to Task 1
Allocated to Task 1
Free
Allocated to Task 2
Allocated to Task 2
Free
Free
Free

We can notice that the semaphore is being allocated to one and another task. Each task makes use of the serial feature by voice, releasing the semaphore when it no longer needs the serial, thus allowing the other task (or itself) to be able to allocate it in the future.

At a certain moment (in the second execution of task 2) the traffic light tries to be allocated, but it is already done, which makes the task that was last allocated to be locked until the semaphore is released.

Note: Allocate and release the semaphore can block the task indefinitely (as in this example), or until a certain timeout has been defined by the programmer, it is necessary to treat an error return on the call to the function that it takes the traffic light, deciding what to do next.

In addition to these mutex (mutex) traffic lights or binary semaphores, we have the counters' traffic lights, which allow different tasks and points of the code to allocate the semaphore, decreasing a value until it reaches zero, when it will no longer be possible to allocate it . The deallocation operations, or release, increment this counter again. Along the libre will be presented examples of the two types of traffic lights, clarifying in a practical way its operation.

Critical Section

A statement in C is most often equivalent to several instructions in machine language. This way when we use access to global variables, in our tasks we have a case in which it can not be guaranteed that the task goes out of context with the statement in half and another task takes the variable with an inconsistent value.

Any operation that needs to be done from start to finish, without the current task changing to outa, can be done within a protected code section, where the switching mechanism between tasks is deactivated (and will be activated only at the end of the block). This block is what we call the critical section.

The critical section starts with a special operating system instruction followed by the instruction (or instructions) that need to be protected from concurrency in order to end the critical section with another OS instruction at the end. In the case of freeRTOS it is called a pair of macros.

software *timer*

In addition to code that runs in tasks, we can use functions that are executed from time to time, without an internal loop from the timer module. The timer module puts the defined functions into a queue executes them at regular intervals or once, after a timing.

Thus, to flash an LED or control a timeout you can activate a call of a certain function every n seconds, for example. If you see that you no longer have to wait for it (why the information you expect arrived, for example), you just need to call an RTOS function to cancel the configuration of this function.

Chapter 3 - Working with Tasks

Using the examples that come with the library

Let's now return to that sample source that is available in the arduino IDE in the menu path: File \ Examples \ FreeRTOS \ Blink_AnalogRead.

The source will not be listed here entirely, but rather some of its excerpts will be highlighted, and even then without some comments. You can search for them by sequencing in your IDE by familiarizing yourself with parts of the Arduino program with freeRTOS. Then we will examine another example and more detailed explanations will be given. Line references are meant to help, but nothing guarantees that your IDE example will not make any difference to mine when I opened this file.

Including the freeRTOS standard header for arduino

This is a standard, minimal include command for using freeRTOS on arduino. It has the most basic definitions, which allow the creation and operation of tasks.

It is the first line of the file:

```
01: #include <Arduino_FreeRTOS.h>
```

If other resources are used, such as semaphores, the inclusion of other header files will be required, depending on each situation.

Task's function prototype

In order to be referenced in the setup () function, which creates tasks, its prototypes (signatures) are defined as follows:

```
03: // define two tasks for Blink & AnalogRead

04: void TaskBlink( void *pvParameters );
05: void TaskAnalogRead( void *pvParameters );
```

Your names should be the ones that best reflect your goal. Always return void and get a generic pointer to void (which can be adapted to the type of data that one wants to pass to the task, as we will see later).

Creation of tasks

In the setup function, the two tasks are created, using the xTaskCreate function of freeRTOS. Each function has some parameters, and by hour only the first one is highlighted, which is the name of the task's function. Thus freeRTOS creates all the controls of the tasks and prepares the system to execute the function passed by parameters like the code of the task:

```
// Now set up two tasks to run independently.
  xTaskCreate(
    TaskBlink
    ,  (const portCHAR *)"Blink"
    ,  128
    ,  NULL
    ,  2
    ,  NULL );

  xTaskCreate(
    TaskAnalogRead
    ,  (const portCHAR *) "AnalogRead"
    ,  128  // Stack size
    ,  NULL
    ,  1  // Priority
    ,  NULL );
```

In bold are the passage of the function with the code of the task.

Empty loop function

You can check that the loop function, which is where you could expect to find code that flashes the LED and reads an analog input, is empty. This is precisely because now the operations will be preferentially in the tasks and when necessary in the interruptions.

This function has nothing to show, in my opinion, above all the paradigm shift of this type of program [1] .

Tasks Functions

The following are the implementations of the functions, omitting comment blocks:

```
46: void TaskBlink(void *pvParameters)  // This is a task.
47: {
48:   (void) pvParameters;
49:
74:    // initialize digital LED_BUILTIN on pin 13 as an output.
75:   pinMode(LED_BUILTIN, OUTPUT);
76:
77:   for (;;) // A Task shall never return or exit.
78:   {
79:     digitalWrite(LED_BUILTIN, HIGH);
80:     vTaskDelay( 1000 / portTICK_PERIOD_MS );
81:     digitalWrite(LED_BUILTIN, LOW);
82:     vTaskDelay( 1000 / portTICK_PERIOD_MS );
83:   }
84: }
```

In line 48, a reference to the parameter, which does nothing in fact, serves to eliminate an alert that the parameter is not used. You can not delete it, since the signature should always be this, this being one of the solutions to satisfy the liker.

At line 75, we have a pin initialization. An interesting thing is that we can do it in the first lines of the task, since this pin is her subject and only she will use it. This code runs only once, before the task enters its eternal loop, at line 77.

Between lines 77 and 83 we have a loop that flashes the LED. First turn on the LED attached to the pin, wait a little, turn it off and then back on. This is the possible implementation for what you intended to implement without using an

RTOS in the blinkLED () of Listing 1 - Example of a C monotask application for microcontroller .

Let us now examine the second task function:

```
086: void TaskAnalogRead(void *pvParameters)  // This is a task.

087: {

088:   (void) pvParameters;


099: for (;;)

100: {

101:     // read the input on analog pin 0:

102:     int sensorValue = analogRead(A0);

103:     // print out the value you read:

104:     Serial.println(sensorValue);

105:     vTaskDelay(1);

106:   }
107: }
```

In line 102 we have a reading of the analog input A0 for the variable sensorValue that is allocated inside the loop.

In line 104 this value is sent by serial. If you compare the sending of values by the serial here with the code that will be shown in the Listing 3: AnalogRead_DigitalRead (example library without some comments) , you will notice that here it was not necessary to use a semaphore to coordinate the exclusive use of the serial simply because only a task sends things by serial.

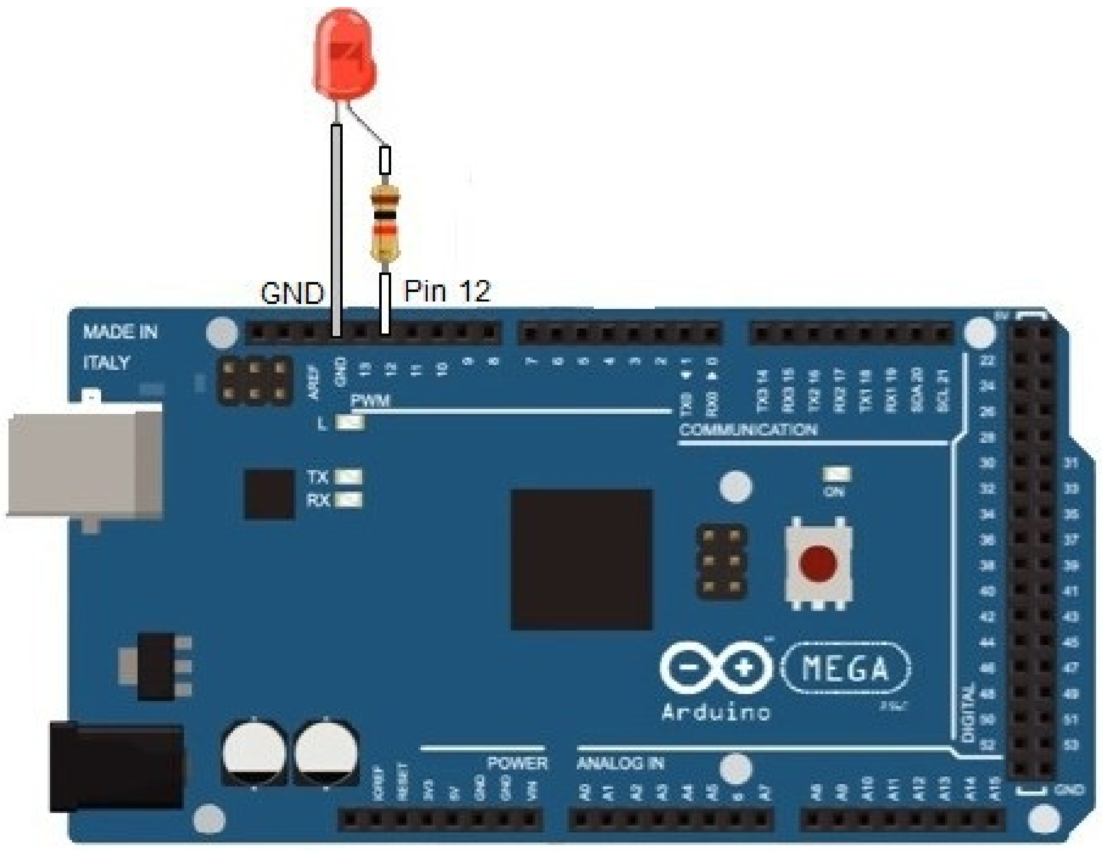
If you use a feature in only one task you can use it quietly, but if you use it in several places you need to control access to it.

Like the `TaskBlink()` function, this task function has a loop. However, in this case, no initialization was necessary.

You may want to run this example on your arduino and watch the serial monitor send the A0 input values (even if you have not connected any sensors to it). If you look for the LED on the Arduino board (no matter the model there is this LED on the Arduino board itself, connected internally to the output 13), you will see that it is blinking as intended.

FreeRTOS itself is concerned with sharing the CPU between the codes of the two functions, alternating them constantly. Try copying the `TaskBlink()` function to another function, called TaskBlinkSaida12, for example, inserting its prototype before the setup () function and creating another task with the same code as the other but with the pin 12 being blinked.

If the idea sounds good to you, use an LED with the cathode connected to GND and a 10kΩ resistor connecting the anode to pin 12 as follows:



Note:

GitHub

This modified code can be found in github. At https://github.com/maxback/multitarefa_na_pratica , as well as some supplementary codes from the book, which should be flagged with this github image.

A second example

To reinforce the basic concepts of the skeleton of the programs, let's look at another example. It is recommended, especially at the beginning, to use ready-made skeletons as the basis for new projects, saving time and avoiding banal mistakes, like forgetting an include or miss signing a task function, mistakenly remembering how it should be written .

Open a second sample source, also available in the Arduino IDE in the menu path: File \ Examples \ FreeRTOS \ AnalogRead_DigitalRead.

The open file has several explanatory comments. Some of these comments have been deleted, resulting in a relatively long code, which follows.

Listing 3: AnalogRead_DigitalRead (example library without some comments)

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>  // add the FreeRTOS functions for Semaphores (or Flags).

SemaphoreHandle_t xSerialSemaphore;

// define two Tasks for DigitalRead & AnalogRead
void TaskDigitalRead( void *pvParameters );
void TaskAnalogRead( void *pvParameters );

// the setup function runs once when[...]
void setup() {

   // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

  while (!Serial) {
    ;
  }

  if ( xSerialSemaphore == NULL )
  {
    xSerialSemaphore = xSemaphoreCreateMutex();
    if ( ( xSerialSemaphore ) != NULL )
      xSemaphoreGive( ( xSerialSemaphore ) );
  }
```

```

   // Now set up two Tasks to run independently.
  xTaskCreate(
    TaskDigitalRead
  ,  (const portCHAR *)"DigitalRead"  // A name just for humans

  ,  128  // This stack size can be checked & adjusted by reading the Stack Highwater
  ,  NULL
  ,  2  // Priority, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the lowest.
  ,  NULL );

  xTaskCreate(
    TaskAnalogRead
  ,  (const portCHAR *) "AnalogRead"
  ,  128  // Stack size
  ,  NULL
  ,  1  // Priority
  ,  NULL );

   // Now the Task scheduler, which takes over control of scheduling individual Tasks, is automatically started.
}

void loop()
{
   // Empty. Things are done in Tasks.
}

/*--------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*--------------------------------------------------*/

void TaskDigitalRead( void *pvParameters __attribute__((unused)) )  // This is a Task.
{
   // digital pin 2 has a pushbutton attached to it. Give it a name:
  uint8_t pushButton = 2;

   // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);
```

```
63:
64:   for (;;) // A Task shall never return or exit.
65:   {
66:     // read the input pin:
67:     int buttonState = digitalRead(pushButton);
68:
69:     if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 )
== pdTRUE )
70:      {
71:       Serial.println(buttonState);
72:
73:       xSemaphoreGive( xSerialSemaphore );
74:     }
75:
76:     vTaskDelay(1);  // one tick delay (15ms) in between reads
for stability
77:   }
78: }
79:
80: void TaskAnalogRead( void *pvParameters __attribute__((unused))
)  // This is a Task.
81: {
82:
83:   for (;;)
84:   {
85:     // read the input on analog pin 0:
86:     int sensorValue = analogRead(A0);
87:
88:     if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 )
== pdTRUE )
89:     {
90:       Serial.println(sensorValue);
91:
92:       xSemaphoreGive( xSerialSemaphore );
93:     }
94:
95:     vTaskDelay(1);
96:   }
97: }
```

The understanding of this code begins with the identification of the setup () and loop () functions, starting on lines 11 and 47, typical of an Arduino program.

In the `setup ()` function, serial is initialized between lines 14 and 18. So far so good. The interesting thing to follow is the beginning of the configuration of the multitasking system, using the freeRTOS function. Moving forward (to ignore the semaphore configuration for an hour), we find the call to the `xTaskCreate()` function, between lines 28 and 34:

```
28:   xTaskCreate(
29:     TaskDigitalRead
30:   ,  (const portCHAR *)"DigitalRead"   // A name just for humans

31:   ,  128   // This stack size can be checked & adjusted by
reading the Stack Highwater
32:   ,  NULL
33:   ,  2   // Priority, with 3 (configMAX_PRIORITIES - 1) being
the highest, and 0 being the lowest.
34:   ,  NULL );
```

This call is responsible for creating the task named "DigitalRead" (Parameters 2, on line 30), which is a text associated with task that can be used especially for debug, a name only for humans, as the author of the code calls it . The code that will be executed is a typical C function whose reference is passed in parameter 1: `TaskDigitalRead()` .

The other parameters of the call will receive our attention at later times in this book.

The function of the task can be given any name, but usually starts with Task, by convention proposed in the freeRTOS example codes and documentation. It can be found between lines 56 and 78.

Comparing this function with the one presented in Listing 2 - Pseudocode of a task in conjunction with an interrupt , we noticed the presence of an initialization command (line 62: `pinMode(pushButton, INPUT);`) and the infinite loop of the task, between the lines 64 and 77.

This loop is constantly reading the state of a button (line 67) with the arduino's `digitalRead ()` function to then send through the serial (line 71). After this the task waits for some time, calling the freeRTOS delay function called `vTaskDelay()` , at line 76.

There are two very important observations about this delay:

- **Do not** use the arduino's delay () function anymore, but the freeRTOS `vTaskDelay ()` function, because this function works in harmony with the task control, taking advantage of this wait to allow the code of another task to re-execute.
- After all, why keep the code in one place, if we have other tasks to do?

- The parameter of this function is in ticks of the clock of the task keeper, that is, it depends on the configuration of freeRTOS. For example, if the timer tick of the schedule is every 15 ms, this delay will be 15 ms, but may change in other settings. So what to do?
- It's actually simple. FreeRTOS has provided a macro to convert the time in microseconds that we want, in the ideal amount of ticks:

```
76:     vTaskDelay( pdMS_TO_TICKS(1000) );
```

We can also do the calculation by finding out how many milliseconds per tick we have in the system, as follows:

```
76:     vTaskDelay( 1000 / portTICK_PERIOD_MS );
```

(The two examples are for a wait of 1000 ms, or 1 second)

This task then repeats the loop and reads the button and sends it to the serial port, voltand while waiting before executing the next time.

The second task has its code in the `TaskAnalogRead ()` function, and has a similar structure, but with a different purpose, since it reads one of the analog inputs of the arduino and sends by the serial, also from time to time. It is created in the `setup ()` function, also using the `xTaskCreate()` function (line 36).

Another detail, which is highlighted by the code author in comment line 44, is that after the setup the task scheduler starts automatically and the tasks created will start automatically. This will always happen unless you do something about it, which is to suspend the task even before it starts. This topic will be discussed later in this book.

Because the two tasks have different priorities (the fourth parameter of the `xTaskCreate()` function), the `TaskDigitalRead ()` task will have a higher execution priority than the `TaskAnalogRead()` task, since its priority is 2 (line 33), while the other has priority 1 (line 41).

In freeRTOS a higher numeric value corresponds to higher priority , unlike other priority systems (such as hardware interrupts usually have, for example).

Note: This is a good time for you to save your project and then run it on your arduino. Open the serial monitor and check the uploads of the digital and analogue input values.
The digital input will contain values between 0 and 1 (you would need to connect input 2, time to 5V, time to gnd, to guarantee the value of your choice). The analogue input will contain values from 0 to 255 (in the same way you can set up a circuit with a potentiometer, for example, and connect the analogue input A0).
However, hardware connections are not critical to understanding this example.

Passing parameters to the task in creation

We can have one task as code for multiple executions, or instances. In this way you can have, for example three tasks with the same code, but executing different operations. For this to be possible the task receives by parameter, in the function parameter, the data needed for its differentiated execution. This value can be from an integer and a string to a data structure (as will be shown in the example in Listing 4).

The definition of the structure, the new tasks, the passing of parameters and "receiving them" will be highlighted and will be explained later.

Listing 4: AnalogRead_DigitalReadModifiedVariousReaders

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>  // add the FreeRTOS functions for Semaphores
(or Flags).



// type of parameters

typedef struct {

  const char *pcTaskName; // name of the task TO BE CREATED

  const char *pcText; // text to be sent by the serial before
reading, identifying its source

  int SensorID; // ID of analog input used

  int DelayTicks; // time between sending of readings, in ticks

} AnalogReadParam_t;


// Declare a mutex Semaphore Handle which we will use to manage the
Serial Port.

// It will be used to ensure only only one Task is accessing this
```

```
resource at any time.

SemaphoreHandle_t xSerialSemaphore;


// define two Tasks for DigitalRead & AnalogRead

void TaskDigitalRead( void *pvParameters );
```

`//accepts as parameter AnalogReadParam_t`

`void TaskAnalogReadParam( void *pvParameters );`

```
// define parameters for three analog read tasks that accept struct
parameters
```

`AnalogReadParam_t xParams[3], *pxParam = xParams;`

```
// the setup function runs once when you press reset or power the
board

void setup() {


  // initialize serial communication at 9600 bits per second:

  Serial.begin(9600);



  while (!Serial) {

    ; // wait for serial port to connect. Needed for native USB, on
LEONARDO, MICRO, YUN, and other 32u4 based boards.

  }


  // Semaphores are useful to stop a Task proceeding, where it
should be paused to wait,
  // because it is sharing a resource, such as the Serial port.

  // Semaphores should only be used whilst the scheduler is running,
but we can set it up here.

  if ( xSerialSemaphore == NULL )  // Check to confirm that the
Serial Semaphore has not already been created.

  {

    xSerialSemaphore = xSemaphoreCreateMutex();  // Create a mutex
semaphore we will use to manage the Serial Port

    if ( ( xSerialSemaphore ) != NULL )

      xSemaphoreGive( ( xSerialSemaphore ) );  // Make the Serial
Port available for use, by "Giving" the Semaphore.

  }


  // Now set up two Tasks to run independently.

  xTaskCreate(

    TaskDigitalRead

    ,  (const portCHAR *)"DigitalRead"  // A name just for humans

    ,  128  // This stack size can be checked & adjusted by reading
the Stack Highwater

    ,  NULL

    ,  (configMAX_PRIORITIES - 3)  // Priority, with 3
(configMAX_PRIORITIES - 1) being the highest, and 0 being the
lowest.

    ,  NULL );




    // define parameters for three analog read tasks that accept
struct parameters
```

```
   pxParam->pcTaskName = "TaskInputA1";

   pxParam->pcText = "Input A1";

   pxParam->SensorID = A1;

   pxParam->DelayTicks = pdMS_TO_TICKS( 250UL );

   pxParam++;


   pxParam->pcTaskName = "TaskInputA2";

   pxParam->pcText = "Input A2";

   pxParam->SensorID = A2;

   pxParam->DelayTicks = 1; //pdMS_TO_TICKS( 250UL );

   pxParam++;


   pxParam->pcTaskName = "TaskInputA3";

   pxParam->pcText = "Input A3";

   pxParam->SensorID = A3;

   pxParam->DelayTicks = 1; //pdMS_TO_TICKS( 250UL );

   pxParam++;



   //xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)&xParams[0], 1, NULL );

   //xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)&xParams[1], 1, NULL );

   //xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128, (
void *)&xParams[2], 1, NULL );
```

```
   // create the tasks (it goes through array of pointers until
arriving at null, to facilitate)

   for(pxParam = &xParams[0]; pxParam < &xParams[3]; pxParam++)

   {

    Serial.println("\n---------------------------------");

    Serial.print("Creating ");

    Serial.println(pxParam->pcTaskName);



    xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)pxParam, (configMAX_PRIORITIES - 2)  /* Priority  */, NULL
);

   }




// Now the Task scheduler, which takes over control of scheduling
individual Tasks, is automatically started.

}


void loop()

{

  // Empty. Things are done in Tasks.

}


void SerialDebugWithSemaphore(const char *pszText)

{

   if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) ==
pdTRUE )
    {
      Serial.print(pszText);

      xSemaphoreGive( xSerialSemaphore ); // Now free or "Give" the
Serial Port for others.
    }
}

/*--------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*--------------------------------------------------*/

void TaskDigitalRead( void *pvParameters __attribute__((unused)) )
  // This is a Task.
{
  /*
    DigitalReadSerial
    Reads a digital input on pin 2, prints the result to the serial
monitor

    This example code is in the public domain.
  */

  // digital pin 2 has a pushbutton attached to it. Give it a name:
  uint8_t pushButton = 2;
  // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);

  for (;;) // A Task shall never return or exit.
  {
    // read the input pin:
    int buttonState = digitalRead(pushButton);

    // See if we can obtain or "Take" the Serial Semaphore.
    // If the semaphore is not available, wait 5 ticks of the
Scheduler to see if it becomes free.
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) ==
pdTRUE )
    {
      // Send the received text by parameter
      Serial.print("digital input: ");

      // We were able to obtain or "Take" the semaphore and can now
access the shared resource.
      // We want to have the Serial Port for us alone, as it takes
some time to print,
      // so we don't want it getting stolen during the middle of a
conversion.
      // print out the state of the button:
      Serial.println(buttonState);

      xSemaphoreGive( xSerialSemaphore ); // Now free or "Give" the
Serial Port for others.
```

```
    }

    vTaskDelay(1);  // one tick delay (15ms) in between reads for
stability
  }
}


void TaskAnalogReadParam( void *pvParameters )  // This is a Task.
{
  // taking parameter as string text
  AnalogReadParam_t *pxParams = (AnalogReadParam_t *) pvParameters;

  for (;;)
  {
    // read the input on analog pin:
    int sensorValue = analogRead(pxParams->SensorID);

    // See if we can obtain or "Take" the Serial Semaphore.
    // If the semaphore is not available, wait 5 ticks of the
Scheduler to see if it becomes free.
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) ==
pdTRUE )
    {
      // Send the received text by parameter
      Serial.print(pxParams->pcText);
      Serial.print(": ");
```

```
     // We were able to obtain or "Take" the semaphore and can now
access the shared resource.

     // We want to have the Serial Port for us alone, as it takes
some time to print,

     // so we don't want it getting stolen during the middle of a
conversion.

     // print out the value you read:

     Serial.println(sensorValue);


     xSemaphoreGive( xSerialSemaphore ); // Now free or "Give" the
Serial Port for others.


    }


   vTaskDelay(pxParams->DelayTicks);  // one tick delay (15ms) in
between reads for stability

  }

}
```

First we define the type `AnalogReadParam_t` , which is nothing more than a structure with data of the name of the task, the text to be sent by the serial before the reading, the identifier of the sensor from where the task should read, as well as the time of delay within each task. We can group our information into types and pass a variable of this type to the task, allowing you to configure the behavior of each specific task.

```
// type of parameters

typedef struct {

  const char *pcTaskName; // name of the task TO BE CREATED

  const char *pcText; // text to be sent by the serial before
reading, identifying its source
  int SensorID; // ID of analog input used
  int DelayTicks; // time between sending of readings, in ticks
} AnalogReadParam_t;
```

The following is the prototype of the task function. Tasks must always have the same signature, so their parameter can not be of type `AnalogReadParam_t` , but of type void *. This is not really a problem, because when instantiating the task you can pass the pointer to the structure by casting `void *` and reading, within the function, cast to the appropriate type (in this case `AnalogReadParam_t *`):

```
// accepts as parameter AnalogReadParam_t
void TaskAnalogReadParam( void *pvParameters );
```

Next we declare an array of 3 positions of the type of the parameter, as well as a pointer to go through this array.

```
// define parameters for three analog read tasks that accept struct
parameters
AnalogReadParam_t xParams[3], *pxParam = xParams;
```

Within the `Setup()` function each element of this array will be initialized and then in a loop tasks will be created with only suitable parameters. Because it deals with purely language operations, we will go directly to the creation of tasks, which is the most interesting:

```
xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128, (void
*)pxParam, (configMAX_PRIORITIES - 2)  /* Priority  */ , NULL );
```

The parameter data is used to name the task itself by passing `pxParam->pcTaskName` , while the integer parameter is passed by pointer and cast to `void *` in `(void *)pxParam` ..

It should always be noted that the task's function is always `TaskAnalogReadParam()` . It is as if the set between the `TaskAnalogReadParam()` function name and the past parameters (`pxParam`) were a class that is instantiated in as many objects as necessary.

Finally, we have the function of the task, which was a modification of the `TaskAnalogRead()` function, of the previous listing, but depending on the parameter for everything (from deciding the text to be sent, to the origin of the reading and the delay time between the readings). It is up to the programmer to know what is different for each instance and what is common to all. Next we will review the receipt of the parameter:

```
AnalogReadParam_t *pxParams = (AnalogReadParam_t *) pvParameters;
```

We have the declaration of a local variable (`AnalogReadParam_t *pxParams`), which points to the variable passed by parameter, with the appropriate type cast: `(AnalogReadParam_t *) pvParameters` .

Important: Each task must have its data, to prevent competition between memory regions. We recommend that they be initialized and then read. If any data should be received or sent, RTOS mechanisms, such as traffic lights and queues, should be used.

Run this code on the arduino and see that on the serial monitor will appear the readings of the three analog inputs A1, A2 and A3:

```
---------------------------------
Creating TaskInputA1

---------------------------------
Creating TaskInputA2

---------------------------------
Creating TaskInputA3
Input A1: 379
Input A2: 324
Input A3: 275
Input A2: 297
Input A3: 294
Input A2: 308
Input A3: 300
Input A2: 291
Input A3: 272
Input A2: 262
Input A3: 258
Input A2: 267
Input A3: 278
Input A2: 281
Input A3: 276
Input A2: 262
digital input: 0
Input A1: 285
Input A3: 262
```

```
    .
    Input A2: 251
    Input A3: 266
```

You may notice that the task that reads input A1 executes at a much larger interval, in the image above it is circled by rectangle. This is because the DelayTicks parameter is initialized with ticks equivalent to 250ms while the others are only 1 tick (equivalent to the arduino ATMega2560 at 15ms). If you match only intervals you will see that the three also send the same amount of readings.

The option to create the tasks in a loop was to demonstrate that you can have some form of dynamic decision (not just fixed in the code), reading (from the EEPROM for example) a configuration and deciding how many tasks you would read the values and what values. The more traditional option was commented out, and can replace the for loop, simply uncomment the following three lines (and exclude or comment the for):

```
xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128, (void
*)&xParams[0], 1, NULL );

   xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128, (void
*)&xParams[1], 1, NULL );

  xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128, (void
*)&xParams[2], 1, NULL );
```

Stopping and restarting task execution

A created task will begin to run automatically when the task schedule is started (which occurs after finishing the Arduino Setup () function).

If you want the task to be suspended, stopped without doing anything, you can suspend it, If this is done before the end of the Setup () function, it will not even start execution. If any event or timing triggers this suspend action, it will be running until this occurs.

You need to use the vTaskSuspend (TaskHandle_t pxTaskToSuspend) function;

It does, however, expect an identifier of the task, which we do not have in our example, since it is returned by the function that creates the task in its last parameter. This way you can change the code of the previous listing to save the last created task, declaring a variable for it, saving the value of each task in the loop and after it suspends the last task, since we always save the last one. The change would look something like this:

```
       // stores the last created task
       TaskHandle_t xLastTask;
       // create the tasks (it goes through array of pointers until
arriving at null, to facilitate)
      for(pxParam = &xParams[0]; pxParam < &xParams[3]; pxParam++)
      {
         Serial.println("\n--------------------------------");
         Serial.print("Creating ");
         Serial.println(pxParam->pcTaskName);

         xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)pxParam, (configMAX_PRIORITIES - 2)  /* Priority  */,
&xLastTask );
      }

       vTaskSuspend(xLastTask);
```

If you run the program you will see that the task that reads from A3 does not run:

```
   --------------------------------
   Creating TaskInputA1

   --------------------------------
   Creating TaskInputA2

   --------------------------------
```

```
Creating TaskInputA3
Input A2: 380
Input A1: 321
Input A2: 341
digital input: 0
Input A2: 346
Input A2: 351
digital input: 0
Input A2: 344
Input A2: 320
digital input: 0
Input A2: 331
Input A2: 316
digital input: 0
Input A2: 305

Input A1: 289
Input A3: 278
Input A2: 292
```

The passage of the handle as a parameter occurs similarly to the structure:

```
,  (void *)xLastTask // Pass the task hadle value with cast to (void *)
```

Now to make the task run again we will have some event done, calling the `void vTaskResume( TaskHandle_t pxTaskToResume ); function.`

For this we need to have the handle of the task in the place where we will suspend it. One idea is to have this in the taks parameter and set to one of them, which will be responsible for initiating it.

The task chosen is the one that reads the digital input (`TaskDigitalRead()`), to start the task when its input becomes worth 1, which can be obtained by connecting the VCC to digital input 2.

However, we will need to put the creation of it after the creation of the tasks that read the digital inputs and pass the task identifier by parameter to it:

```
    .
    .
    .

         xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)pxParam, (configMAX_PRIORITIES - 2)  /* Priority  */ ,
&xLastTask );
      }

      vTaskSuspend(xLastTask) ;
```

```
        // Now set up two Tasks to run independently.
      xTaskCreate(
        TaskDigitalRead
        ,  (const portCHAR *)"DigitalRead"  // A name just for
humans
        ,  128  // This stack size can be checked & adjusted by
reading the Stack Highwater
        ,  (void *)xLastTask //Pass the task hadle value with cast
to (void *)
        ,  (configMAX_PRIORITIES - 3)  // Priority, with 3
(configMAX_PRIORITIES - 1) being the highest, and 0 being the
lowest.
        ,  NULL );
```

The digital input task, with the required changes in bold is shown below:

```
    void TaskDigitalRead( void *pvParameters)  // This is a Task.
   {
      /*
      DigitalReadSerial
      Reads a digital input on pin 2, prints the result to the
serial monitor

      This example code is in the public domain.
     */

      // digital pin 2 has a pushbutton attached to it. Give it a
name:
     uint8_t pushButton = 2;

      TaskHandle_t xTaskParaReiniciar = (TaskHandle_t) pvParameters;

      // make the pushbutton's pin an input:
     pinMode(pushButton, INPUT);

     for (;;) // A Task shall never return or exit.
     {
        // read the input pin:
      int buttonState = digitalRead(pushButton);

        // See if we can obtain or "Take" the Serial Semaphore.
      // If the semaphore is not available, wait 5 ticks of the
Scheduler to see if it becomes free.
      if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) ==
pdTRUE )
      {
         //Envia o Text recebido por parâmetro
         Serial.print("digital input:");

         // We were able to obtain or "Take" the semaphore and can
now access the shared resource.
         // We want to have the Serial Port for us alone, as it
takes some time to print,
         // so we don't want it getting stolen during the middle of
a conversion.
         // print out the state of the button:
         Serial.println(buttonState);

         xSemaphoreGive( xSerialSemaphore ); // Now free or "Give"
the Serial Port for others.

         // Decide whether to restart task
         if( (xTaskParaReiniciar != NULL) && (buttonState == 1) )
         {
          vTaskResume(xTaskParaReiniciar);
         }

      }

      vTaskDelay(1);  // one tick delay (15ms) in between reads for
stability
     }
   }
```

Notice that the task identifier (or hadle) to be restarted is read from the parameter and when the input is changed to logic level 1 it is reset. The attribute that said that the pvParameters parameter was not used was also dropped (`__attribute__((unused))`).

Although, after the first time the task is restarted, other calls to this operation are ignored as the task is started.

Run the project with these modifications and you will get a response like the following one (the arrow indicating the moment the input goes to one and the other highlights showing that the task has been re-run:

```
digital input: 0
Input A1: 237
Input A2: 230
Input A2: 238
digital input: 0
Input A2: 241
Input A2: 234
```

```
    digital input: 0
    Input A2: 231
    Input A2: 233
    digital input: 0
    Input A2: 246
    Input A2: 238
    digital input: 0
    Input A2: 236
    Input A2: 230
    digital input: 0
    Input A1: 241
    Input A2: 235
    Input A2: 244

    digital
input:  1

    Input A2: 242

    Input  A3:
237
    Input A2: 234

    Input  A3:
231
    Input A2: 230

    Input  A3:
235
    Input A2: 229

    Input  A3:
225
    Input A2: 227
    Input A1: 248

    Input  A3:
240
```

This complete code can be found in Apendix A.1 - AnalogRead_DigitalREadModificadoVariosLeitores (Com suspensão da task) . Use it to compare with your modifications and find any coding errors or just to better examine the complete example.

Chapter 4 - Working with Semaphores

In Listing 3: AnalogRead_DigitalRead (example library without some comments) we have already seen a MUTEX semaphore in action, although at that time the focus was not serial access control itself.

The following is a simpler example, which sends two passed texts per parameter for each of the tasks that have the same function. In this way only the parameter differentiates the execution of one task from the other. We will execute this project and understand how the semaphore works. Then we will deactivate the traffic light and see the result on the serial monitor.

MUTEX Semaphore

Examine the code of our example, which will be explained in the sequence.

GitHub

Listing 5: ExampleMethodMutexSerial

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

SemaphoreHandle_t xSerialSemaphore;

void TaskSendTextToSerial( void *pvParameters );

void setup()
{
   Serial.begin(9600);
   while (!Serial) ;


   if ( xSerialSemaphore == NULL )  // Check to confirm that the Serial Semaphore has not already been created.
   {
     xSerialSemaphore = xSemaphoreCreateMutex();  // Create a mutex semaphore we will use to manage the Serial Port
     if ( ( xSerialSemaphore ) != NULL )
       xSemaphoreGive( ( xSerialSemaphore ) );  // Make the Serial Port available for use, by "Giving" the Semaphore.
   }

   if ( xSerialSemaphore != NULL )
   {
      // Create two tasks with the same function, passing different texts by parameter
     xTaskCreate(TaskSendTextToSerial, "TaskSendTextToSerial1", 128, (void *)"Text 1", 1, NULL );
     xTaskCreate(TaskSendTextToSerial, "TaskSendTextToSerial2", 128, (void *)"Text 2", 1, NULL );
   }
27:    else
28:    {
29:      Serial.println("**** Error creating semaphore!");
30:    }
31: }
32:
33: void loop()
34: {
35:   // Nothing in the loop. All in the tasks!
36: }
37:
38: void TaskSendTextToSerial( void *pvParameters )
39: {
40:    // taking parameter as string text
41:   const char *pszText = (const char *) pvParameters;
42:
43:   for (;;)
44:   {
45:     // Suggestion to change: Switch portMAX_DELAY by
(TickType_t) 5 to ticks and then fold
46:     if ( xSemaphoreTake( xSerialSemaphore, portMAX_DELAY ) ==
pdTRUE )
47:     {
48:       Serial.println("-------------------------");
49:       Serial.println(pszText);
50:
51:       xSemaphoreGive( xSerialSemaphore );
52:
53:     }
54:     vTaskDelay(pdMS_TO_TICKS(1000));
55:   }
56: }
57:
58:
```

Let's explain: In line 2, we have the `semphr.h` , file include, which defines the type and functions of semaphores.

In line 4, in the global scope of the program, we have the `xSerialSemaphore` semaphore declaration, of type `SemaphoreHandle_t` . This type is always the same regardless of the type of semaphore that will actually be created.

Next, in the `setup()` function, we have a check of the value of the `xSerialSemaphore` variable, at line 14, which is actually unnecessary, since we know that it was not created outside `setup()` . Then the function responsible for creating a mutex-type semaphore is called: `xSemaphoreCreateMutex()` . If you return a value other than null in `xSerialSemaphore` , it means that it was created successfully. (usually due to the lack of available memory, which clearly is not our case in this simple context).

The correct creation of the semaphore is tested on line 17, to then call the function that frees the semaphore:

```
16:   xSerialSemaphore = xSemaphoreCreateMutex();  // Create a mutex semaphore we will use to manage the Serial Port
17:   if ( ( xSerialSemaphore ) != NULL )
18:     xSemaphoreGive( ( xSerialSemaphore ) );  // Make the Serial Port available for use, by "Giving" the Semaphore.
```

The call to `xSemaphoreGive` causes the semaphore to be freed, and can be picked up (allocated) by the first task requesting it. At this point we do not have the tasks created yet. This will be done on lines 24 and 25 that create two tasks with the same `TaskSendTextToSerial()` function, but passing different text parameters:

```
(void *)"Text 1"
(void *)"Text 2"
```

After `setup()` the two tasks start the wheels and we can see the following result in the serial monitor:

```
-------------------------
Text 1
-------------------------
Text 2
-------------------------
Text 1
-------------------------
Text 2
-------------------------
Text 1
-------------------------
Text 2
```

As the semaphore gets released, the first task that requests it, with the `xSemaphoreTake()` function, will succeed, as can be seen on line 46 :

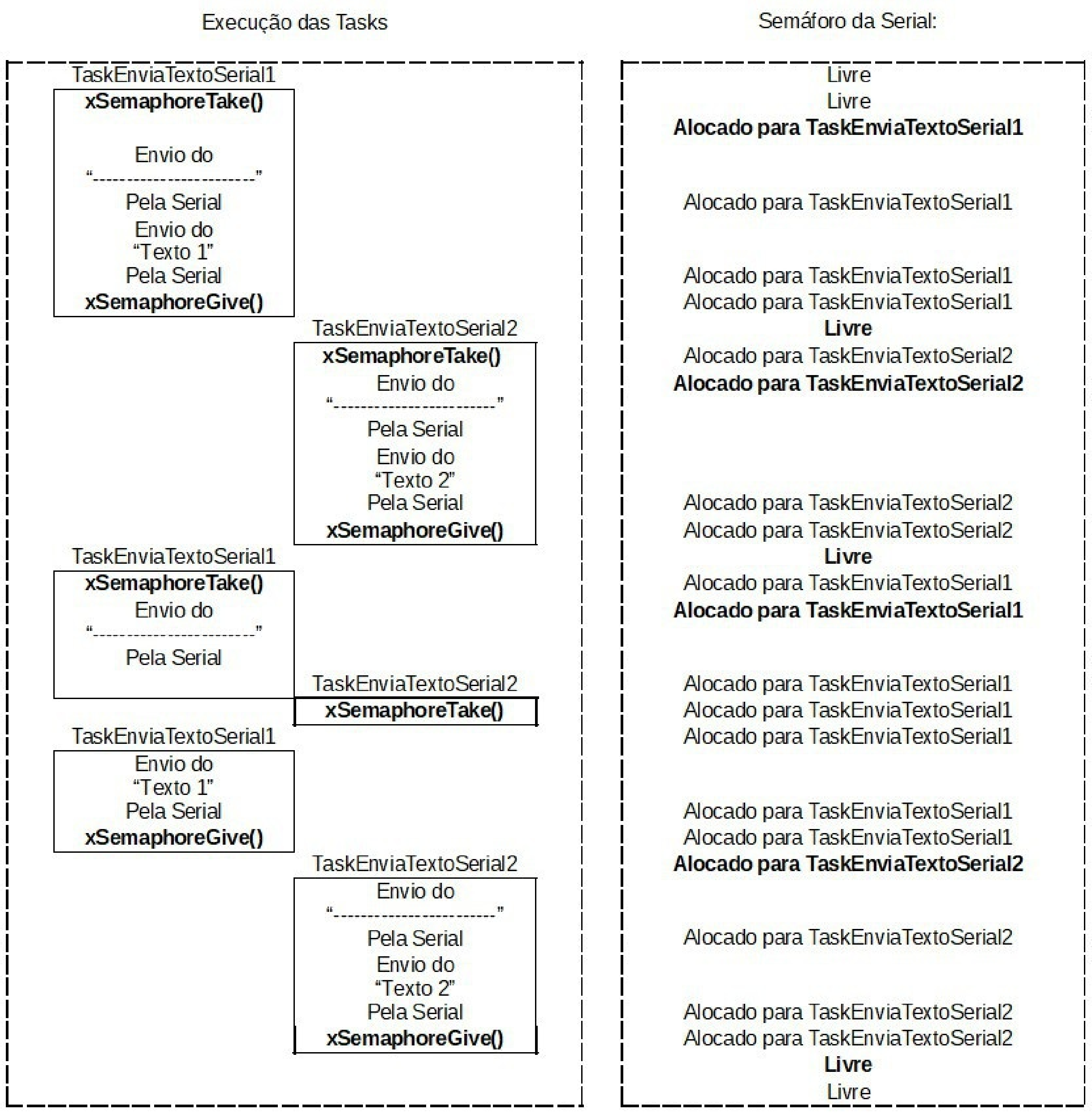
```
    46:         if ( xSemaphoreTake( xSerialSemaphore, portMAX_DELAY )
== pdTRUE )
```

Once the semaphore is allocated, and thus access to the Serial object, a sequence of 25 traces is sent followed by a line break character, followed by the text received per parameter itself.

When the two calls to `Serial.println()` conclude the semaphore is released by calling the `xSemaphoreGive()` function, which frees the semaphore, at line 51. In this way, the other task that was “caught” in the call to `xSemaphoreTake()` will finally get the semaphore and you may have the Serial object just for it. Releasing it then.

It waits indefinitely because the second parameter of the xSemaphoreTake function is the wait time in ticks (the first is the semaphore variable, which is global). If we pass the constant `portMAX_DELAY` instead of a tick value, such as 5, it will stay indefinitely on that line until another task that is locking the semaphore releases it.

This sequence can be seen in the figure below:


Execução das Tasks
Semáforo da Serial:
TaskEnviaTextoSerial1
xSemaphoreTake()
Envio do
"------------------------"
Pela Serial
Envio do
"Texto 1"
Pela Serial
xSemaphoreGive()
TaskEnviaTextoSerial2
xSemaphoreTake()
Envio do
"------------------------"
Pela Serial
Envio do
"Texto 2"
Pela Serial
xSemaphoreGive()
TaskEnviaTextoSerial1
xSemaphoreTake()
Envio do
"------------------------"
Pela Serial
TaskEnviaTextoSerial2
xSemaphoreTake()
TaskEnviaTextoSerial1
Envio do
"Texto 1"
Pela Serial
xSemaphoreGive()
TaskEnviaTextoSerial2
Envio do
"------------------------"
Pela Serial
Envio do
"Texto 2"
Pela Serial
xSemaphoreGive()
Livre
Livre
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Livre
Alocado para TaskEnviaTextoSerial2
Alocado para TaskEnviaTextoSerial2
Alocado para TaskEnviaTextoSerial2
Alocado para TaskEnviaTextoSerial2
Livre
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial1
Alocado para TaskEnviaTextoSerial2
Alocado para TaskEnviaTextoSerial2
Alocado para TaskEnviaTextoSerial2
Alocado para TaskEnviaTextoSerial2
Livre
Livre

Now we will dispense with the use of the semaphore, without controlling simultaneous calls to the main method () of the Serial object and see what happens. Just comment lines 46, 51 and 54. Commenting on line 54, removing the call to `vTaskDelay()` is intended to intensify the dispute over the serial and increase the chances of problems. The result should be similar to the following:

```
43:    for (;;)
44:    {
```

```
45:        // Suggestion to change: Switch portMAX_DELAY by
(TickType_t) 5 to ticks and then fold
46:       //if ( xSemaphoreTake( xSerialSemaphore, portMAX_DELAY ) ==
pdTRUE )
47:      {
48:        Serial.println("------------------------");
49:        Serial.println(pszText);
50:
51:         //xSemaphoreGive( xSerialSemaphore );
52:
53:      }
54:       //vTaskDelay(pdMS_TO_TICKS(1000));
55:    }
```

Run your project and you will see on the serial monitor that the sending of the characters is scrambled, taking part part of the text sent by the first test, time of the second:

```
----------------
Text 1
----------------
---t 2
----------------
Text 1
----------------
---t 2
----------------
Text 1
----------------
---t 2
----------------
Text 1
----------------
---t 2
----------------
Text 1
----------------
---t 2
```

This semaphore not only controls the access of two tasks, but several. Try uncommenting the control of the semaphore and adding three more tasks, adding the new creation calls after line 25, changing only the text sent to the task and its name:

```
    xTaskCreate(TaskSendTextToSerial, "TaskSendTextToSerial3",
128, (void *)"Text 3", 1, NULL );
    xTaskCreate(TaskSendTextToSerial, "TaskSendTextToSerial4",
128, (void *)"Text 4", 1, NULL );
    xTaskCreate(TaskSendTextToSerial, "TaskSendTextToSerial5",
128, (void *)"Text 5", 1, NULL );
```

Você deverá obter o seguinte resultado:

```
-------------------------
Text 1
-------------------------
Text 2
-------------------------
Text 3
-------------------------
Text 4
-------------------------
Text 5
-------------------------
Text 1
-------------------------
Text 2
-------------------------
Text 3
-------------------------
Text 4
-------------------------
Text 5
-------------------------
Text 1
```

Binary Semaphore

Unlike a MUTEX semaphore, which protects a resource from being accessed by more than one point of the code, causing a mutual exclusion, the binary semaphore functions as a flag. Obviously this flag is not a simple flag, but rather it has all the features of a real-time operating system and with scheduling of tasks. It works with 1 bit, as the name says, however this bit can be set from another task or from an interrupt and then tested (or expected) on a task.

To better understand this, we will make a comparison between the solution with flags bits and the binary semaphore in an RTOS system.

Testing bits to handle events

Without using an operating system, programming the firmware directly to the hardware, we usually use bit flags to warn the loop in the main function (or some function called it) that an event has occurred, usually from an interrupt. This also applies to arduino programs, which use interrupts in some cases to have greater hardware control.

Let's look at an example where you want to monitor a pin attached to a key and warn the computer (via serial) of each occurrence of change in the pin goes to one, indicating that the key has been switched on.

Listing 6: InterruptionKeyPinFlagParaLoop (without RTOS)

```
const byte KeyPin = 2;
volatile bool flag_keyOn = false;

void setup() {
   Serial.begin(9600);

   pinMode(KeyPin, INPUT_PULLUP);
   attachInterrupt(digitalPinToInterrupt(KeyPin),
     handleKeyPinInterrupt, RISING);
}

void loop() {
   if(flag_keyOn)
```

```
  {
    flag_keyOn = false;
    Serial.print("Key on!");
  }

   // Various system activities would be performed here, for example!
}

void handleKeyPinInterrupt() {
  flag_keyOn = true;
}
```

In this example, we have a flag marked as `volatile` called `flag_keyOn` , which is marked as true in the function that is bound to the interrupt of uplink (from logic level 0 to 1) of pin 2, where our key is attached.

Then in the `loop()` function the flag is constantly tested, and when it is true it is immediately zerad (false) and the text sent by the serial. Once marked false, we have the option of capturing an upcoming event in the `EventChangeInterface ()` as soon as it occurs.

The comment "`// Various system activities would be executed here, for example!` " Simulates the point at which other system operations are executed, regardless of key control and sending to the PC.

Using a binary semaphore to handle events

In our RTOS we use binary semaphores for this purpose, so that we can have all the benefits of being able to associate a task to handle the key (or set of keys), in a more flexible way without having to fill its flags program in the interruptions and tests in the program loop. One can then stay with the task set waiting for the traffic light to signal that the event was indicated, then processing the information. If the task is the one with the highest priority, it will be immediately executed, even if another task was running before the interruption, and was given a lot of time to handle the event more adequately.

We will then see the version the use of a binary semaphore and the considerations of its use and difference in relation to the MUTEX semaphore.

Listing 7: InterruptionKeyPinParaTask (using RTOS)

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

const byte KeyPin = 2;
////////////// Flag deleted !! ///////////////
//volatile bool flag_keyOn = false;


 // Binary semaphore to indicate key event
SemaphoreHandle_t xBinarySemaphoreKey;

// Task to notify the key bind
void TaskKey( void *pvParameters );
// Simulate other things the program needs to do constantly
void TaskOtherSystemActivities( void *pvParameters );


void setup() {

   BaseType_t xRet;

   Serial.begin(9600);

   pinMode(KeyPin, INPUT_PULLUP);

    // Create the binary semaphore
   xBinarySemaphoreKey = xSemaphoreCreateBinary();

    // If all is ok, create the task
   if(xBinarySemaphoreKey != NULL) {

      xTaskCreate( TaskOtherSystemActivities,
         (const portCHAR *) "OtherSysAct", 128, NULL, 1 , NULL);

      xRet = xTaskCreate(
       TaskKey
       ,  (const portCHAR *) "TaskKey"
       ,  128  // Stack size
       ,  NULL
       ,   2  // Priority
       ,  NULL );

       if(xRet == pdPASS)
       {
         attachInterrupt(digitalPinToInterrupt(KeyPin),
           handleKeyPinInterrupt, RISING);
       }
       else
       {
         Serial.println("*** Error creating task!");
       }
   }
   else
   {
      Serial.println("*** Error creating semaphore!");
   }

}

void loop() {
   // Flag monitoring left here
}

void handleKeyPinInterrupt() {
   // Variable to control the task change after leaving the interrupt.
   // This scheme should always be done to use the function that terminal in FromISR
   BaseType_t xHigherPriorityTaskWoken;

   // Uses the semaphore to indicate key event. Unlocking the task waiting for him.
   // Pass the address of xHigherPriorityTaskWoken to then treat the scheduler before exiting
   // the interruption.

   xSemaphoreGiveFromISR( xBinarySemaphoreKey, &xHigherPriorityTaskWoken );

```

```
075:    // This is the call that should be made at the end of ISR () for freeRTOS to change task,
076:    // if applicable
077:   if( xHigherPriorityTaskWoken )
078:   {
079:     taskYIELD ();
080:   }
081: }
082:
083: // Task to notify the key on
084: void TaskKey( void *pvParameters ) {
085:    // infinite task loop
086:   for( ;; )
087:   {
088:      // Wait for the signal to be triggered
089:     xSemaphoreTake( xBinarySemaphoreKey, portMAX_DELAY );
090:      // Send by serial information
091:     Serial.print("Key on!");
092:   }
093:
094: }
095:
096: // Simulate other things the program needs to do constantly
097: void TaskOtherSystemActivities( void *pvParameters ) {
098:
099:    // Infinite Loop
100:   for(;;) {
101:      // Various system activities would be performed here, for example!
102:   }
103: }
```

In line 6, the statement `flag_keyOn` flag is commented to show that this type of mechanism is no longer used, but the semaphore whose global variable is defined in line 10:

```
010: SemaphoreHandle_t xBinarySemaphoreKey;
```

Note:

Note: You may notice that the type of the variable is the same as the MUTEX semaphore, which makes things more flexible, to pass a semaphore to a function that manipulates it, without knowing exactly the real type of it (in orientation the objects this is associated to polymorphism). The creation function that initializes the data within the type structure appropriately to each type, remaining the `SemaphoreHandle` type. Imagine that you want to hide the details a bit or create a more descriptive name function in Portuguese. You could have the following signature:

```
TakeSemaphore(SemaphoreHandle_t xBinarySemaphoreKey);
```

or else:

```
GetSemaphore(SemaphoreHandle_t xBinarySemaphoreKey);
```

To talk about two possibilities ... The function does not really have to know what kind of semaphore, because this operation can be done on all types of semaphore with different meanings, but calling the same function of the freeRTOS: `xSemaphoreTake ()` API.

Next we have two tasks tasks: `TaskKey()` and `TaskOtherSystemActivities()` on lines 13 and 15, the first one being the dedicated task in monitoring on handling key changes through the binary semaphore.

The second task has the function of exemplifying the other operations of the system, which vary according to its purpose. They override the same simulation done in the previous listing in the loop function, but now with a task.

The `setup()` function has been changed to create the binary semaphore after the pin configuration, at line 27:

```
027:    xBinarySemaphoreKey = xSemaphoreCreateBinary();
```

Note that the `xSemaphoreCreateBinary ()` function is used instead of the `xSemaphoreCreateMutex ()` function, so that the binary semaphore type is created. At line 30 the variable is tested. If it is null it means that some error happened (which is reported in the "else" of lines 53 to 56, sending a text by serial). The predicted error is basically out of memory, in dynamic memory allocation (which is not expected in the allocation at the beginning of the software, unless you have already allocated all the memory in some giant statically declared array, for example).

Then with the successfully created semaphore, tasks will be created on lines

32 and 35. The return of success was only implemented for the second tasks, but the most robust solution is to test the return of the creation of both taks. This check is made for the second tasks (which the key handles) on lines 48 through 51. Note that the return tests consist of comparing a BaseType_t variable with `pdPASS` . `pdPASS` is a constant used to return this function executed successfully.

Note: Each adaptation of freeRTOS for a specific platform defines in type `BaseType_t` as the type that matches the size of the word on that platform. So if your platform is 8 bits this type will be 8 bits. The same goes for 16, 32 and 64 bits.

Using this type causes the operations to be done with the type in which they will be faster. We should use these types to prevent problems of different types.

Then, on line 64, the function `handleKeyPinInterrupt()` that treats the external interrupt in the key pin, as before, but now adapted to use the semaphore created. Here's a crucial point: By checking the key and sending signaling with the semaphore it has been linked, we expect that the tasks involved in this information will be warned as soon as possible. After the event has been notified (with the semaphore) the task waiting for it is again able to be executed and can execute after the interruption if it has more priority than the one that is running, which is our case.

We can see the event being flagged by calling the `xSemaphoreGiveFromISR()` function at line 73. Note that the function executed is special for interrupts (ends in FromISR). The variable `xHigherPriorityTaskWoken` , comes with the freeRTOS information about whether to change the task. The freeRTOS system needs a little help within the interrupt to switch tasks. It is used at line 77, calling the function (or C macro) `taskYIELD ()` , if the variable is true. This function, as already mentioned, changes the execution controls to execute the next task, just before leaving the interrupt.

In line 84 we finally have the task waiting for the signal semaphore at the interrupt (`xBinarySemaphoreKey`). The function waits indefinitely for the semaphore (since it passes the parameter `portMAX_DELAY` as timeout) on line 89, since the `xSemaphoreTake ()` function blocks the execution of this task xSemaphoreTake until the semaphore. Note that the task does not even run from time to time, since freeRTOS, knowing that it waits for the semaphore, temporarily removes it from the list of tasks that can be executed from time to time.

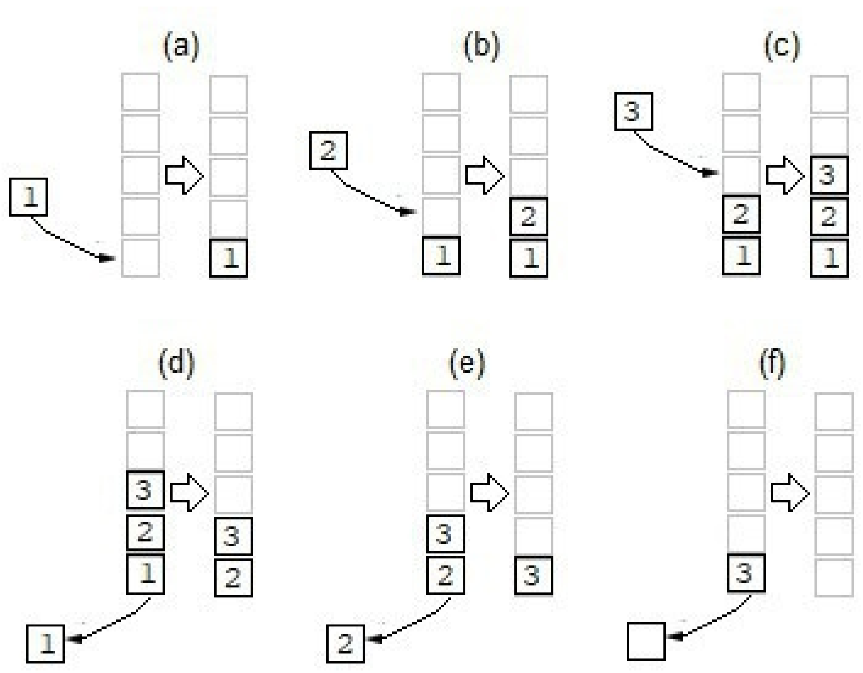
We add the task `TaskOtherSystemActivities()` (line 97), to show the other things that the system would be doing while the key is not triggered. This pro in a real program you would have this and other tasks, each taking care of your subjects. You can imagine this key as the alloy key of your smartphone, which responds with a menu that allows, among other things, to turn off the cellphone.

Chapter 5 - Data Queues

Communication between tasks

When we need to communicate multiple items between parts of our code we usually pass arrays by parameters or use global variables and access between the two points.

Another mechanism that can be used is a queue, with one or more positions that makes the communication synchronized between tasks or interruptions and tasks. Each time we visualize better how this can be done let's imagine that we have a queue of numbers and that a task is putting numbers in sequence. First it puts the number 1, then the 2 and finally the number 3. The second task is consuming these items, that is, it from time to time picks up items from the queue, if they exist. For the sake of simplicity let's consider that the second tasks only begin to remove the items after the first have entered all three [2] .



Note Each step indicates the before and after step, separator by an arrow from left to right .

Steps:

(a) The task1 adds in the empty queue the item 1;

(b) The task1 adds in the queue that already is with item 1 item 2;

(c) The task1 adds in the queue that already is with items 1 and 2 item 3;

(d) task2 removes item 1 from the queue leaving it with two items (values 2 and 3);

(e) Task2 removes item 2 from the queue leaving it with item 3;

(e) Task2 removes item 2 from the queue leaving it with item 3;
(f) Task2 removes item 3 from the queue leaving it empty.

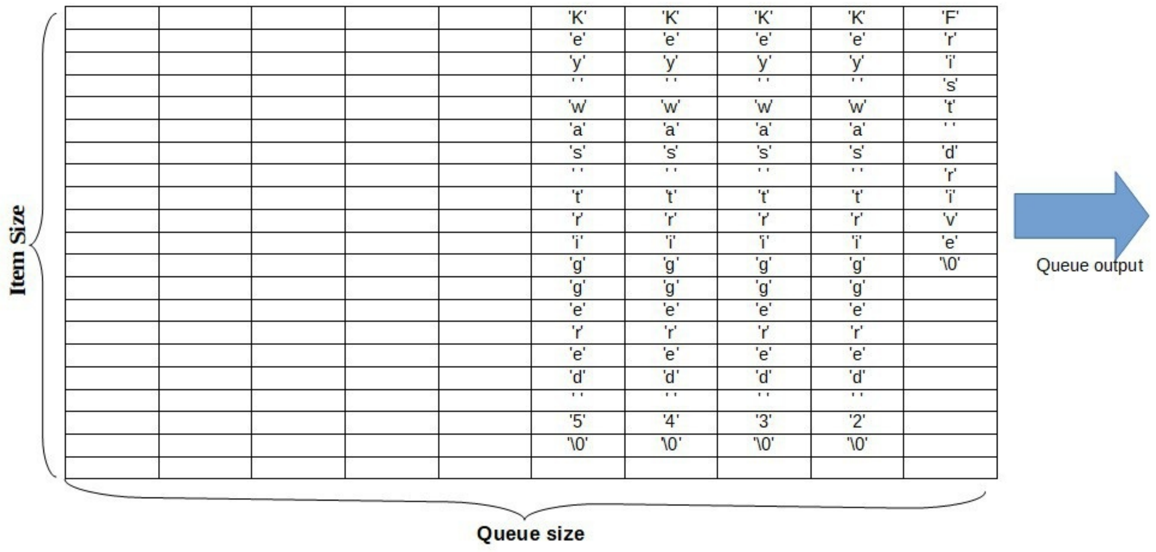
Let's now take a practical example!

Our first example queued code will consist of an interrupt function and a task, the first one that monitors a key (based on Listing 7: InterruptionKeyPinParaTask (using RTOS)) and each drive will send to the task a text with a message to be sent to the serial (and viewed on the Arduino's serial monitor).

Before we present the code, there are some comments about our queue. It will have a number of items (ten), each item being the size of 21 bytes, since we can put the type of data we want in the queue (in the example of the figure was a numeric value and here a string). This size of 21 bytes is enough to put a text with up to twenty characters (because a byte must be reserved for the null character of text end). So you can put texts like the following:

```
First drive
Key triggeded 2
Key triggeded 3
Key triggeded 4
Key triggeded 5
```

Here's an illustration of how these items would look in the queue:


Item Size
Queue output
Queue size
'F' 'r' 'i' 's' 't' ' ' 'd' 'r' 'i' 'v' 'e' '\0'
'K' 'e' 'y' ' ' 'w' 'a' 's' ' ' 't' 'r' 'i' 'g' 'g' 'e' 'r' 'e' 'd' ' ' '2' '\0'
'K' 'e' 'y' ' ' 'w' 'a' 's' ' ' 't' 'r' 'i' 'g' 'g' 'e' 'r' 'e' 'd' ' ' '3' '\0'
'K' 'e' 'y' ' ' 'w' 'a' 's' ' ' 't' 'r' 'i' 'g' 'g' 'e' 'r' 'e' 'd' ' ' '4' '\0'
'K' 'e' 'y' ' ' 'w' 'a' 's' ' ' 't' 'r' 'i' 'g' 'g' 'e' 'r' 'e' 'd' ' ' '5' '\0'

To adopt this example, we will create a new program similar to Listing 7: InterruptionKeyPinParaTask (using RTOS) , but using the queue in place of the semaphore. So we can highlight only the part of the queue, since the rest is already understood. Follow the code:

Listing 8: FilaTextSendKey

```
#include <Arduino_FreeRTOS.h>
#include <queue.h>
const byte KeyPin = 2;

// Queue identifier
QueueHandle_t xQueue;

char szTextfFirstItem[] = "First drive";
char szTextOfOtherItems[] = "Key was triggered X";
char nCounter = 0;

// Task to notify the key bind
void TaskSend( void *pvParameters );


void setup() {

  BaseType_t xRet;

  Serial.begin(9600);
  attachInterrupt(digitalPinToInterrupt(KeyPin),
handleKeyPinInterrupt, RISING);
  pinMode(KeyPin, INPUT_PULLUP);

   // Creates the character submission queue
   xQueue = xQueueCreate(
     10,                   // Queue size: 10
     21 *sizeof( char )   // Item Size: 21
   );


  if(xQueue) Serial.println("Queue created successfully!");

  xRet = xTaskCreate(TaskSend, (const portCHAR *) "TaskSend",
 128, NULL, 2, NULL);
```

```

  if(xRet == pdPASS) Serial.println("Task created successfully!");
}

void loop() {
  // Flag monitoring left here
}


void handleKeyPinInterrupt() {
  BaseType_t xHigherPriorityTaskWoken;

  nCounter++;
  if(nCounter > 9) nCounter = 2;

  if(nCounter == 1)
  {
    // places the item in the queue passing the specific text to the first item
    xQueueSendToBackFromISR( xQueue, (void *)szTextfFirstItem, &xHigherPriorityTaskWoken);
  }

  else
  {
    // Mounts the text to be copied to the queue item
    szTextOfOtherItems[18] = '0' + nCounter;

    // place the item in the queue passing the text to be copied
    xQueueSendToBackFromISR( xQueue, &szTextOfOtherItems, &xHigherPriorityTaskWoken);
  }

  if( xHigherPriorityTaskWoken )
  {
    taskYIELD ();
  }
}

// Task to notify the key bind
void TaskSend( void *pvParameters ) {
  // Stored text to be received
```

```
074:   char szText[21];
075:
076:   Serial.println("Starting loop: TaskSend");
077:
078:     // loop of the task
079:   for( ;; )
080:   {
081:      // Wait indefinitely until you read an item in the queue
082:      xQueueReceive( xQueue, &szText, portMAX_DELAY );
083:
084:     Serial.print(szText);
085:   }
086: }
```

The queue variable is declared global form on line 7 with the type QueueHandle_t. The creation of the queue itself is done in lines 26 to 29:

```
026:   xQueue = xQueueCreate(
027:     10,                  // Queue size: 10
028:     21 *sizeof( char )   // item SIZE: 21
029:   );
```

In line 26, the `xQueueCreate ()` function is called with two parameters to create the queue, in the first parameter we pass the queue size (in the case 10 items) and in the second parameter we pass the size, in bytes, of each item. We used the sizeof operator to get the number of char bytes and then multiplied by 21.

The variable `xQueue` receives the reference to the queue that will be used both in the interrupt that sends data to the queue and in the task that reads this data. In line 32 this variable is tested, and if it has a true value it is signaled by the serial [3] .

Let's now use the queue:

Between lines 44 and 69 we have the function `handleKeyPinInterrupt()` , which implements the service of treatment of the external interruption associated to the pin of our key. In our case, this function is called at each key trigger to place a text item in the queue.

Note: Note that manipulating text is not the best approach in an interrupt, as it should be as short as possible, but will serve a first example, which will then be modified to also at this point.

Well! we initialize three global variables that are only used by this interrupt (which eliminates competition for data): `szTextfFirstItem`, `szTextOfOtherItems e nCounter`. The first two are used to determine the text to be sent (the first one is fixed on line 53 and the second one is edited before sending on line 59). The variable nCounter is used to differentiate the first sending of the others and to determine the number for the second message onwards.

Note: For the sake of simplicity of code, after message 9 everything restarts.

At line 53, the first call of data is sent to the queue, placing the item at its end:

```
053: xQueueSendToBackFromISR( xQueue, (void *)szTextfFirstItem,
&xHigherPriorityTaskWoken);
```

The first parameter is the queue identifier, `xQueue`, is the queue identifier. It tells the function which queue, among possible queues, should receive the data.

The second parameter is the generic pointer to the item. In our case, we expect a vector of bytes (characters) of 21 positions. Note that you always pass a pointer to the memory region that must be passed, but the queue has a memory area for the item of the specified size (in this case 21 bytes).

What happens is that 21 bytes are copied from the memory of the vector to the memory of the item. Generic pointer casting (`void *`) is not always necessary, but in our case it shows its nature.

The third parameter is proper of the RTOS API functions and serves for context switching (like the semaphore function in Listing 7) and in our example it is used at line 65 to decide to call the function to force the scheduler's evaluation .

Note: The `xQueueSendToBackFromISR()` function is a special function for interrupts (terminates in FromISR). If we are sending an item from a task, just call the version of the `xQueueSendToBack()` function (which will be exemplified in the next example).

Once the texts are put in the queue, someone has to take care of them. This is done by the function of the created task, which is between lines 72 and 86. Like tasks seen so far, it has an infinite loop called the queue reading function and the action to be performed when any item is received:

```
081:        // Wait indefinitely until you read an item in the queue
082:        xQueueReceive( xQueue, &szText, portMAX_DELAY );
083:
```

```
084:      Serial.print(szText);
```

At line 82 we have to call the `xQueueReceive ()` function, which reads the queue item. The first parameter is the identifier of the queue and in the second we must pass a pointer to the variable where a copy of the data of the item should be placed, in our case the text to be sent by the serial, with size compatible with that of the item. In this case we always need to pass the address, that is, precede the variable with the & operator.

In the third parameter, which indicates how long (in ticks) will wait for an item, we pass the constant `portMAX_DELAY` so that the function stays indefinitely on that line, until the interrupt places at least one item in the queue. This approach causes the task to wait, waiting to handle a new item in the queue.

Exercise:

The following improvements and applications are suggested as exercises:
- Let's add a task between the interrupt and the task sending the text;
- The interruption will no longer treat the text and will be shorter;
- Let's measure the approximate time between one press and another;

(In Annex 2 you can see a possible solution to this exercise, but I encourage you to at least try to put together the knowledge you have gained so far and be able to resolve this exercise)

- To do this you should:
  - Create a new queue with the size of the queue element to the size of a value that saves the time in ticks: `sizeof (TickType_t)` .
- Change the interrupt to read the tick counter with the function `xTaskGetTickCountFromISR ()` and send to the queue and create an intermediate task to handle these values that creates a variable to know the value of the last counter received and another to pass by pointer to the reception function, both of type `TickType_t` . The time can be calculated as follows:

```
nTempoEmMs = (xNow – xTimeofLastInterrupt) *
portTICK_PERIOD_MS;
```

- This intermediate task should then, using sprintf () or another similar function, write something like "Button: 2000 ms" in the original queue (assuming you left 2 seconds between one press and another.) Note that the second task does not need no fit, as long as the item size with 21 characters is sufficient In this case you should use the normal send function:

```
xQueueSendToBack(xQueue, szText, portMAX);
```

Test your program by noting that the time is sent very accurately, since the tick counter resolution is very high. The first reading will show the time from the beginning of the execution and the others will show the time between readings.

Sending structures from various sources

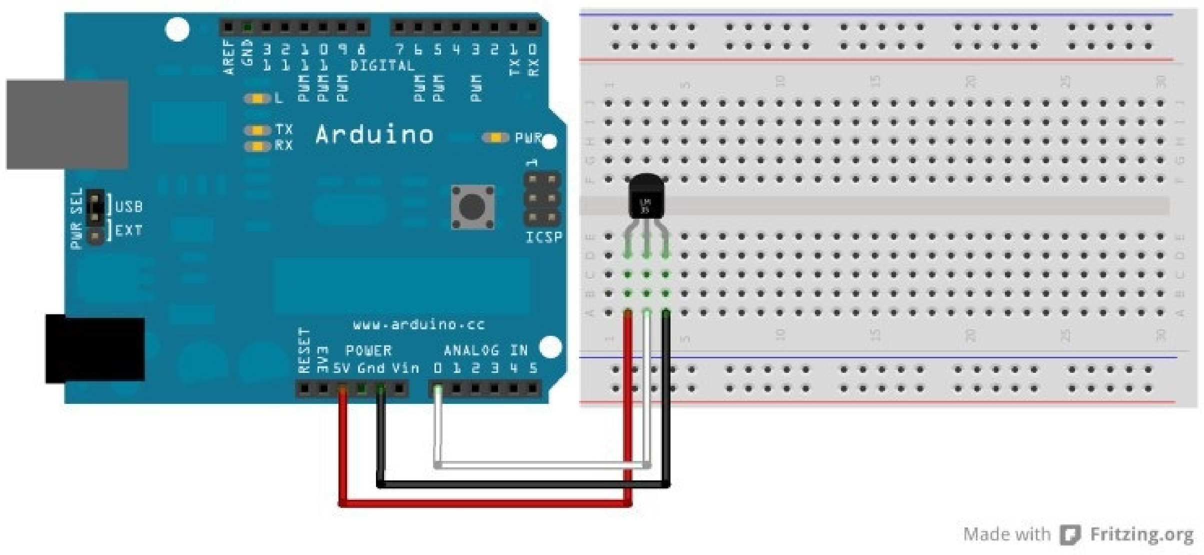
To make things more interesting we will work with a queue that has items being added from more than one source, that is, we will have a task reading the items in the queue and more than one task putting values in the queue.

It is true that the next example is not such a practical application, but suppose we have an LCD or TFT display that shows data coming from more than one source:

- A temperature sensor connected to the analog input;
- A simulated internal clock with timings of a task (second to second);
- The last text received by the serial;

To implement this you can use the LM35 sensor and follow the tutorial elaborated by Alan Motta and available at https://portal.vidadesilicio.com.br/lm35-medindo-Temperature-com-arduino (Released on 07/17/2018) .

Advancing things a little, he rides the following circuit:



Source: https://portal.vidadesilicio.com.br/lm35-medindo-Temperature-com-arduino

As it happened before, early in the book, you do not need to assemble the circuit (but it would be really cool if you did) and leave the door open or connect a potentiometer between VCC and GND with the third terminal connected to analog port 0, simulating the temperature sensor. The task responsible for handling this sensor will read its values every 500 ms and queue the read information, already converted to degrees centigrade.

For the second source we will create a task that is stopped by a following and then calculates the approximate time and sends to the queue.

Already for the third source our task will read the serial 0 (default) and pick up each line received limiting the text to a certain size.

The last serial task to communicate with the display. We will not actually implement this in this example, but rather send two lines with 16 characters each at each change in the data, simulating the update of a 2x16 LCD. It is an exercise to implement hardware and communication with the LCD.

Before we present the listing, it may be interesting to tell you what the queue item will look like:

So far we've been working with text and a numeric value. For sending data from several sources, we will use a structure with one item identifying the source (and thus the type of information) and another one defined by a union (to interpret a common memory region in different ways). The fonts were defined by an enumerated type:

```
   // Enumerated type that indicates what information it has in the
queue item
    typedef enum {itTemperatureReading, itHourry, itReceivedText }
ItemType_t;
```

And the structure by another type, which will be used for local variables in the tasks, function parameters and definition of the size of the items in the queue:

```
   // Type for all three data types (use union) to interpret the
memory in each way
   typedef struct {
    ItemType_t Type;
    union {
     float Temperature;
     struct {
        uint8_t Hour;
        uint8_t Minutes;
        uint8_t Seconds;
     } Horary;
     char Text[13];
    } Data;
   } QueueItem_t;
```

The `QueueItem_t` type will be seen throughout the code and now you already know how to understand it beforehand.

Follow the listing, examine it, looking at the various tasks, ancillary functions, and using the queue. To provide the queue references will be highlighted, so you can easily observe the points where items are inserted at the end of the queue and where items are removed.

Listing 9: ReadingTemperatureLogText

```
#include < Arduino_FreeRTOS.h>
#include <queue.h>

// Enumerated type that indicates what information it has in the queue item
typedef enum {itTemperatureReading, itHourry, itReceivedText} ItemType_t;

// Type for all three data types (use union) to interpret memory in every way
typedef struct {
  ItemType_t Type;
  union {
    float Temperature;
    struct {
      uint8_t Hour;
      uint8_t Minutes;
      uint8_t Seconds;
    } Horary;
    char Text[13];
  } Data;
} QueueItem_t;

// Single row identifier
QueueHandle_t xQueue ;

void TaskReadTemp( void *pvParameters );
void TaskCurrentTime( void *pvParameters );
void TaskSerialReception( void *pvParameters );
void TaskSend( void *pvParameters );

int createTasks(void)
{
```

```
{
    if(xTaskCreate(TaskReadTemp, (const portCHAR *) "TaskReadTemp",  128, NULL, 2, NULL) == pdPASS)
      Serial.println("Task TaskReadTemp was created successfully!");
    else
      return 0;

    if(xTaskCreate(TaskCurrentTime, (const portCHAR *) "TaskCurrentTime",  128, NULL, 2, NULL) == pdPASS)

      Serial.println("Task TaskHour created successfully!");
    else
      return 0;

    if(xTaskCreate(TaskSerialReception, (const portCHAR *) "TaskSerialReception",  128, NULL, 2, NULL) == pdPASS)
      Serial.println("Task TaskReceptionSerial created successfully!");
    else
      return 0;

    if(xTaskCreate(TaskSend, (const portCHAR *) "TaskSend",
      128, NULL, 2, NULL) == pdPASS)
      Serial.println("Task Task Ship successfully created!");
    else
      return 0;

    return 1;
}

void setup() {
  BaseType_t xRet;

  Serial.begin(9600);

  // Create queue for temperature readings
  xQueue = xQueueCreate(10, sizeof( QueueItem_t ));
  if( xQueue )
  {
    Serial.println("Queue created successfully!");
    if(!createTasks())
    {
      Serial.println("Error creating tasks!");
    }
  }
  else
  {
    Serial.println("There was an error creating the queue!");
  }
}

void loop() {
  // Flag monitoring left here
}


void TaskReadTemp( void *pvParameters )
{
  QueueItem_t Item;
  Item.Type = itTemperatureReading;

  Serial.println("Starting loop: TaskReadTemp");

  // loop of the task
  for( ;; )
  {
    int sensorValue = analogRead(A0);

    Item.Data.Temperature = (float(sensorValue)*5/(1023))/0.01;

    xQueueSendToBack( xQueue , &Item, portMAX_DELAY);

    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

void treatClock(QueueItem_t *pItem)
{
    pItem->Data.Horary.Seconds++;
    if(pItem->Data.Horary.Seconds == 60)
    {
      pItem->Data.Horary.Seconds = 0;
      pItem->Data.Horary.Minutes++;

      if(pItem->Data.Horary.Minutes == 60)
      {
        pItem->Data.Horary.Minutes = 0;
        pItem->Data.Horary.Hour++;
        if(pItem->Data.Horary.Hour == 24)
        {
          pItem->Data.Horary.Hour = 0;
        }
      }
    }
}


void TaskCurrentTime( void *pvParameters )
{
  QueueItem_t Item;
  Item.Type = itHourry;
  Item.Data.Horary.Hour = 0;
  Item.Data.Horary.Minutes = 0 ;
  Item.Data.Horary.Seconds = 0;

  Serial.println("Starting loop: TaskSerialReception");

  // loop of the task
  for( ;; )
  {
    treatClock(&Item);

    xQueueSendToBack( xQueue , &Item, portMAX_DELAY);

    //wait a second
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}

void TaskSerialReception( void *pvParameters )
{
  QueueItem_t Item;
  char readedChar;

  int size = 0;

  Item.Type = itReceivedText;
  Item.Data.Text[size] = '\0';

  Serial.println("Starting loop: TaskSerialReception");

  // loop of the task
  for( ;; )
  {
    if(Serial.available())
    {
      readedChar = (char)Serial.read();
      if( readedChar != '\n')
      {
        Item.Data.Text[size++] = readedChar;
        Item.Data.Text[size] = '\0' ;
      }

      if( (readedChar == '\n') || (size == (sizeof(Item.Data.Text)-1)) )
      {
        xQueueSendToBack( xQueue , &Item, portMAX_DELAY);

        size = 0;
        Item.Data.Text[size] = '\0';
      }
    }
  }
}


void TaskSend( void *pvParameters ) {
  char DisplayMirror[2][17] = {"00:00:00 |      ",
                               "T: 000.0 |      "};
  QueueItem_t Item;
  int l,i;
  char *p;


  Serial.println("Starting loop: TaskSend");

  // loop of the task
  for( ;; )
  {
    // Wait indefinitely
    xQueueReceive( xQueue , &Item, portMAX_DELAY );

    // according to the type of the item, updates the display mirror and sends via Serial
    switch(Item.Type)
    {
      case itTemperatureReading:
        sprintf(&DisplayMirror[1][3], "%3.1f", Item.Data.Temperature);
      break;

      case itHourry:
        sprintf(&DisplayMirror[0][0], "%0.2d",
206:            Item.Data.Horary.Hour);
207:          sprintf(&DisplayMirror[0][3], "%0.2d",
208:            Item.Data.Horary.Minutes);
209:          sprintf(&DisplayMirror[0][6], "%0.2d",
210:            Item.Data.Horary.Seconds);
211:        break;
212:
213:        case itReceivedText:
214:          // accommodates part of the text in 6 characters at
the top and 6 at the bottom.
215:
216:           //clean
217:
218:          for(l=0; l<2; l++)
219:          {
220:            for(i=0, p = &DisplayMirror[l][10]; i<6; i++, p++)
*p = ' ';
221:            *p = '\0';
222:          }
223:
224:          //copy
225:          for(l=0; l<2; l++)
226:          {
227:            p = &DisplayMirror[l][10];
228:            for(i=0; i<6 && Item.Data.Text[i]; i++)
229:            {
230:              *p = Item.Data.Text[i];
231:              p++;
232:            }
233:
234:            *p = '\0';
235:          }
236:
237:        break;
238:      }
239:
240:
241:      Serial.println("----------------");
242:      Serial.println(DisplayMirror[0]);
243:      Serial.println(DisplayMirror[1]);
244:    }
245: }
```

Ufa! This listing has gotten bigger. But worth it there. Not everything will be commented on to speed things up ...

In the lines between 29 and 54 we have a function to assign the tasks, returned 0 (false) if any of them return error and 1 (true) if all goes right. This function is called the `setup ()` function.

In line 62 the single queue is created and if everything goes right the function that creates the tasks is called in line 66.

Now let's discuss the tasks functions one by one.

Temperature sensor reading task

The first one is the task `TaskReadTemp()` defined from line 82. First, it instantiates a local variable for the item, which will be known only by the task and already initiates the type of the item to be temperature reading (`itTemperatureReading`) in lines 84 and 85 .

Within the task loop a reading of the analogue input A0 is performed on line 92 and the temperature value is calculated and assigned to the item in line 94.

This reading is then added to the queue at line 96:

```
096:     xQueueSendToBack(xQueue, &Item, portMAX_DELAY);
```

The address of the item is passed to the function which then copies all the bytes of the item to the position of the new item in the queue, which then stores it individually. There is no relationship to the data in the queue and the variable instantiated in line 84. The queue holds everything isolated, controlling even the competition, so that it will not occur from the task that reads the items read its value before it is copied and all configured queue controls. It is not necessary to protect this code as a critical section or something, since the API function itself takes care of this. It can happen as soon as the addition of this item is performed, the task waiting for the items to run and the current task is already suspended, especially if the task of reading had a higher priority than this one.

Then a 500 ms delay is performed, making the task suspended during this time, then repeat the whole operation for a new reading.

Note that in this way we have a well-isolated code snippet, which does what it needs and sends it to another point, which it does not even know, a copy of the data that will be treated as needed, without this being changed at this point. (I never tire of noting the beauty of this decoupling).

Current time generation task

The second of the functions is the `TaskCurrentTime()` function defined from line 123. In the same way as the previous task, it instantiates a local variable for the item, which will be known only by the task and already starts the type of the item, this time indicating that it is a time item (`itHourry`).Also starts the item with the time all reset. The control of the current time will be done in the item itself, by calls to `treatClock()` function, spaced in approximately one second on line 136, inside the main loop.

Just a little! What? Can I call a function from within the task function?

Yes. We've done that by calling the freeRTOS API functions and arduino functions (or methods). One must take care of functions that we call and those that we write (as in this case is now our case) is that the function is reentrant, that is, have a code that can be called more than one task at a time, without that the data shuffle.

You will notice that the call is identical to that of line 96, in the previously discussed task. This is due to the fact that the variable of the item has the same name (could be different) and the queue is the same. Remember that we have three production tasks being a consumer of the data. The big difference is in the data sent to the queue, which has a different type and will need to be treated differently from the items in the received temperature or message readings (which we will see below).

In addition the task function is sending from second to second a new value to the queue through the call that adds the new time value to the queue at line 138:

```
13 8:      xQueueSendToBack(xQueue, &Item, portMAX_DELAY);
```

You will notice that the call is identical to that of line 96, in the previously discussed task. This is due to the fact that the variable of the item has the same name (could be different) and the queue is the same. Remember that we have three production tasks being a consumer of the data. The big difference is in the data sent to the queue, which has a different type and will need to be treated differently from the items in the received temperature or message readings (which we will see below).

Serial receive task

The third task is implemented in the Serial `TaskSerialReception()` function defined from line 145.

Between lines 147 and 153 this function creates two variables to control the reception character char (`readedChar` and `size`) and like the others has a variable for the item and tries to define its type (`itReceivedText`) and initialize the data with reception counter with an empty string (lines 152 and 153).

In line 160, check if there is information on the serial object and if it reads and adds to the buffer of the item itself (line 165), if it is not a line break character ('\ n').

At line 169 tests whether the character '\ n' arrived or if the buffer capacity was all used. In either of these cases, add the item to the queue and restart the receive control.

Note that after copying the contents of the item to the queue it can already be used as in line 166, where the null character is placed in the first buffer position `Item.Data.Text` .

Note: This reception is not the greatest wonder in the world, because if the user does not enter as many characters as necessary to fill the buffer or does not configure the terminal to send a '\ n' the item will not be queued, but our purpose already serves [4] .

Task that reads from the queue

Finally!

This task, whose function that implements it is called `TaskSend()` and starts at line 181, has as its heart the reading of the items of the queue at line 195 with the function `xQueueReceive ()` and the treatment of the item from line 198.

I propose that you examine the "compacted" version of this code below:

```
194:     // Wait indefinitely
195:     xQueueReceive( xQueue , &Item, portMAX_DELAY );
196:
197:      // according to the type of the item, updates the display
mirror and sends via Serial
198:     switch(Item.Type)
199:     {
200:       case itTemperatureReading:
201:         sprintf(&DisplayMirror[1][3], "%3.1f",
Item.Data.Temperature);
202:       break;
203:
204:       case itHourry:
205:          // to line 210: Update DisplayMirror with time
211:       break;
212:
213:        case itReceivedText:
214:          // to line 235: Refresh DisplayMirror with text
236:
237:        break;
238:      }
239:
240:
241:      Serial.println("---------------");
242:      Serial.println(DisplayMirror[0]);
243:      Serial.println(DisplayMirror[1]);
```

In line 195 the variable of the item inserted first in the queue (or the oldest one not yet removed) is copied to the Item variable. If there is no item the task is suspended because the wait time for a reading passed in the third parameter was `portMAX_DELAY`.

If we pass another value (5 ticks, for example) we would need to test after the return of the function if `pdPASS` returned, something like:

```
  if(xQueueReceive( xQueue , &Item, 5) == pdPASS)
  {
    // according to the item type, refresh the mirror ...
   switch(Item.Type)
   {
...
```

But this is not our case ...

Returning: The item we read may have been added for each of the three tasks, so to treat it we use the decision structure “switch case”, testing its type and writing a block of code for each expected type.

Once we know the type, we can pick up the union `Item.Data` the appropriate information and use it to update the part of the mirror to the appropriate 2x16 LCD:

```
   If itTemperatureReading will read from Item.Data.Temperature
   If itHourry will read from Item.Data.Horary
   If the itReceivedText will read from Item.Data.Tex t
```

Finally, it will send this mirror to the serial simulating the display update (lines 241 to 143) and wait for the next item again. As this task does not have timings, even having the same priority of the other tasks it will treat the queue quickly, always being close to zero.

As this code is interesting and I want to propose a practical exercise, it is listed again in annex, but in a more compact and without the lines.

Proposed exercises:

• Run the attachment program as it is;

• Create a project with the original sources and make changes and free experiences. If you use the example with the key here and add a new input type, sent by the interrupt, changing a display character with a different character each time the key is triggered?

• Create another project with the sources and try to shorten the timing within the `TaskReadTemp()` function loop of 500 / portTICK_PERIOD_MS to 400 / portTICK_PERIOD_MS, 300 / portTICK_PERIOD_MS, 100, 50 / portTICK_PERIOD_MS, and finally 1 (one tick) and check how the sending the clock data in relation to sending the temperature data (things may get a little strange).

• Change the priorities of the production tasks (in the `createTasks()` function) to a value below (from 2 to 1) and keep that of the consuming task (`TaskSend`) with a value of 2.

In this way the latter will have the highest priority than the others and will take control every time an item is added in the queue. Check how the execution gets.

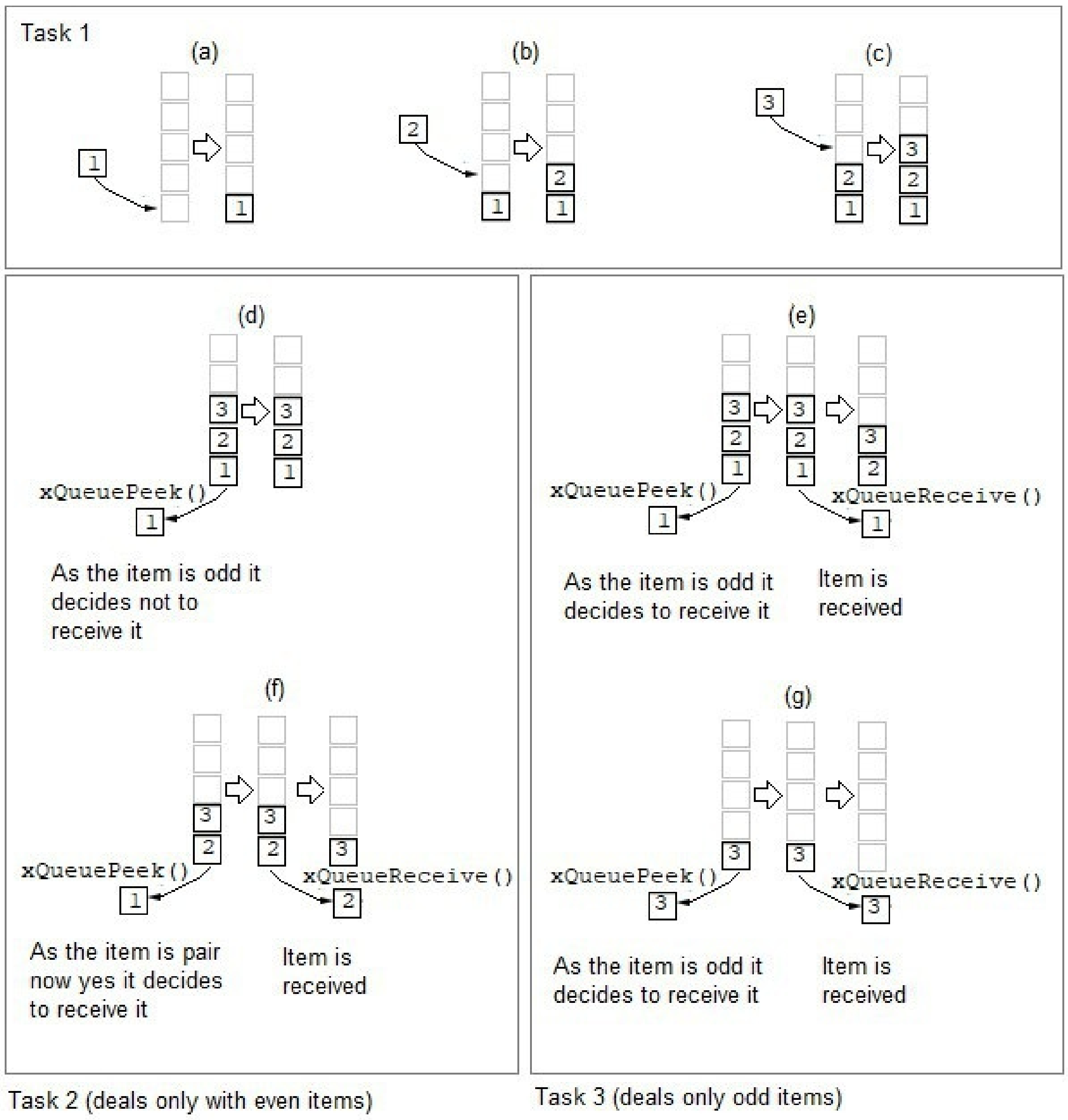
Other options for accessing a queue

So far we have used the `xQueueSendToBack()` function, its variant for interruptions (`xQueueSendToBackFromIRS ()`), both to insert elements at the end of the queue and the `xQueueReceive ()` function to remove items from the queue. However, we have other options of functions in the API of freeRTOS, among which we will exemplify the use of four of them:

- `xQueuePeek()`
- `xQueueOverride()`
- `xQueueSendToFront()`
- `xQueueReset()`

Given a peek at the item without removing it

There are situations where you take a peek at the item and decide whether to pull it out of the queue or not a good idea. You may decide to leave it in the queue (for another task to remove it) if it is not addressed to you, or if you have a data type that you are not able to handle. This is illustrated in the following figure:


Task 1
(a)
1
(b)
2
(c)
3
(d)
xQueuePeek()
As the item is odd it decides not to receive it
(e)
xQueuePeek()
xQueueReceive()
As the item is odd it decides to receive it
Item is received
(f)
xQueuePeek()
xQueueReceive()
As the item is pair now yes it decides to receive it
Item is received
(g)
xQueuePeek()
xQueueReceive()
As the item is odd it decides to receive it
Item is received
Task 2 (deals only with even items)
Task 3 (deals only odd items)

Note: Performing the steps from sa (a) through (g) and each step indicates the before and after the queue in the step, tab by an arrow from left to right.

Steps:

(a) The task1 adds in the empty queue the item 1;

(b) The task1 adds in the queue that already is with item 1 item 2;

(c) The task1 adds in the queue that already is with items 1 and 2 item 3;

(d) Task2 spies the item with `xQueuePeek ()` and "sees" that it is not for it, because it is odd, failing to remove it from the queue.

(e) Task3 spies the item with `xQueuePeek ()` and "sees" that it is up to it to remove the item from the queue because it is odd. Retrieving it with a call to xQueueReceive ().

(f) Task2 spies the item "sees" that it is up to it to remove the item from the queue, since it is even. Removing it with then the queue.

(g) Task3 "sees" that the item is odd, removes it from the queue and makes it empty.

Let's code a bit more ...

To illustrate this we will modify the code in Listing 9, creating another task that will consume the queue, but only for the type of time data, which is no longer sent in the display mirror and is only tested to simulate an alarm system.

We will no longer repeat the whole listing, but the modified `TaskSend()` function and the new `TaskTreatsClock()` function, which corresponds to the new task's code. You should already be able to change the above code to declare the function and create the new task.

Examine the code of the previous function (with the modifications in bold) and the new function and carefully read the explanation that will follow:

```
void TaskSend( void *pvParameters ) {
   char DisplayMirror[2][17] = {"00:00:00 |      ",
                                "T: 000.0 |      "};

   QueueItem_t Item;
   int l,i;
   char *p;
    int treated = 0;

   Serial.println("Starting loop: TaskSend");

    // loop of the task
   for( ;; )
   {
     // Wait indefinitely
      xQueuePeek( xQueue, &Item, portMAX_DELAY );
      treated = 1;
     // according to the type of the item, update the display mirror and send by Serial or ignore the item
     switch(Item.Type)
     {
       case itTemperatureReading:
        sprintf(&DisplayMirror[1][3], "%3.1f", Item.Data.Temperature);
       break;

      case itHourry:
        // Nothing is done except flag item rejected
        treated = 0;
       break;

      case itReceivedText:
        / accommodates part of the text in 6 characters at the top and 6 at the bottom.
        //clean
        for(l=0; l<2; l++)
        {
          for(i=0, p = &DisplayMirror[l][10]; i<6; i++, p++) *p = ' ';
          *p = '\0';
        }

        //copia
        for(l=0; l<2; l++)
        {
          p = &DisplayMirror[l][10];
          for(i=0; i<6 && Item.Data.Text[i]; i++)
          {
            *p = Item.Data.Text[i];
            p++;
          }
          *p = '\0';
        }
      break;

    }

    if( treated)
    {
        // to effectively exit the queue and send testo the serial at this point
        xQueueReceive( xQueue, &Item, portMAX_DELAY );

```

```
         Serial.println("---------------");
         Serial.println(DisplayMirror[0]);
         Serial.println(DisplayMirror[1]);
     }
     else
     {
       // Timed by a tick to induce move to another task
         vTaskDelay(1);
     }
   }
}

void TaskTreatsClock( void *pvParameters ) {
  QueueItem_t Item;

  Serial.println("Starting loop: TaskTreatsClock");
   // loop of the task
  for( ;; )
  {
    // Wait indefinitely
    xQueuePeek( xQueue, &Item, portMAX_DELAY );
     if( Item.Type != itHourry)
    {
       // Timed by a tick to induce to pass ...
        vTaskDelay(1);
      continue;
    }

    xQueueReceive( xQueue, &Item, portMAX_DELAY );
     // Send a new item as text
     // signaling alarms
    if( ( Item.Data.Horary.Hour == 0) &&
        ( Item.Data.Horary.Minutes == 0) &&
        ( Item.Data.Horary.Seconds == 10) )
    {
      Item.Type = itReceivedText;
      sprintf(Item.Data.Text, "Alarme10s !");
    }
    else if( ( Item.Data.Horary.Hour == 0) &&
             ( Item.Data.Horary.Minutes == 2) &&
099:              ( Item.Data.Horary.Seconds == 0) )
100:      {
101:        Item.Type = itReceivedText;
102:        sprintf(Item.Data.Text, "Alarme120s !");
103:      }
104:      else
105:        continue;
106:
107:      xQueueSendToBack(xQueue, &Item, portMAX_DELAY);
108:    }
109: }
```

Before we get the explanation, it is news that this complete source, with the proposed modifications on listing 9 is available from github, in the link: https://github.com/maxback/multitarefa_na_pratica/tree/master/readTemperatraR and also the code that implements the example of even and odd numbers separation in https://github.com/maxback/multitarefa_na_pratica/tree/master/SeparaParesDeIm

Explanation

Since the numbering indications start with 1, we start in line 1 with the pre-existing `TaskSend()` function and addition of the variable used in line 8, which will aim to indicate after each item read from the queue whether it was processed or not. If treated it will be effectively removed from the queue, otherwise it will simply be ignored and the task will not do anything with the item.

Inside of for loop, on line 16, in the moonlight of the receiving function we are accustomed can be seen a call to the `xQueuePeek()` function. This function has the same parameters as the previous one, but with a different behavior: It reads the item but does not exit the queue, allowing its software to decide whether to remove it or leave things as they are.

In line 17 the variable "treated" receives the true value in line 19 we have the switch case from before. If the kind are `itTemperatureReading` and `itHourry` , the item is treated as before. Nothing changed. However at line 25 the case type for the `itReceivedText` received only mark used as false and does not handle the item.

In line 54 the variable used is used to decide to receive the item (and finally remove it from the queue on line 57) and send the display mirror over the serial if used has a true value.

If it is false, the `vTaskDelay ()` function is called to tick a tick and give the opportunity to the other task to execute (only inducing this, since the scheduler can interrupt it due to time-slice).

For this task everything is done, that is: It handles two types of items and leaves the other for another task, which is the task implemented by the `TaskTreatsClock()` function, which starts at line 71.

The function `TaskTreatsClock()` is similar to the previous one, but without a `"switch case"` , since the interest is in only one type, quite the test with if () of line 80, that timing and returns to the next iteration of for loop, using continue, on line 84.

At line 241 the item is then removed from the queue to free up space on it and then evaluated in the tests of lines 244 and 251 which tests the time of 10 seconds and 120 seconds (2 minutes), respectively. Each test of this, if true, edits the item's own variable, "recycling" it for another purpose: Set the item of kind itReceivedText and assemble and copy a text, simulating a reception by the serial to show text indicative of the alarm.

If none of this ifs executes the continue command is called, to pass the next iteration of the loop.

At line 261 the text item is sent to the end of the queue and will be processed by the TaskSend task shortly after processing any items added by the other tasks.

Overwriting an item

In some cases we can use a queue with capacity for just one item to pass information to a task that will have to be dealt with. But what if the task still has not dealt with the information and the event has already been repeated? Another question: What if once you have not been treated it is no longer useful to treat you, and it is more important to treat the new information now?

In these cases the most appropriate treatment is to overwrite the item.

You already sent someone to open a window and it took time to open, and when the window is still going to tea, did you realize that it would be better to open the door? You override one command with the other. Now another situation is when we have a queue of several items and you want to add a new item at the end of it, but if it was not possible you would prefer to overwrite this item to stay with your suspended task waiting to release queue space.

These are two cases where the `xQueueOverride()` function can be used to override an item in a queue. It will be used soon in the book and you can see its use in practice.

Jumping the queue (jumping de line)

So far we work with queues adding the items always at the end of the queue, that is, the good old FIFO system (First In First Out). But for a multitude of situations that may present themselves, you might want to insert the beginning item of the queue, to be dealt with as soon as possible.

I'll cite an example: In the previous listing, which is actually only part of the listing, modified to have one more task, I was tempted to put the new text item in case of alarm at the beginning of the queue, which would be equivalent to replacing the has withdrawn with the time detected as an alarm by another with the text, alerting the user of the alarm as soon as possible.

It would just change the call of line 107 from:

```
xQueueSendToBack(xQueue, &Item, portMAX_DELAY);
```

to:

```
xQueueSendToFront(xQueue, &Item, portMAX_DELAY);
```

This code will cause the alarm text to be processed as soon as possible, since items that were waiting will be moved backwards, and this item will occupy the first position, which will be the next to be taken out of the queue.

Resetting the queue

Another interesting feature is that you can reset the queue, that is, clear the queue if an event occurs, which results in the decision that the data in the queue should no longer be processed, such as pressing a cancel button.

In this case you can simply call the function `xQueueReset ()` , passing the identifier of it. Another example is that of a task that has to read a queue of packets to be sent to a peripheral by the serial and another of the packets captured via DMA or serial reception interrupt. It may occur at any given time what has been received and has not been treated yet to be of no longer interest, and thus you can clear the queue of data received for a new send sequence.

In this case the data queue to send can have mixed packets and commands, and one of these commands is to clear the receive queue ("cmd: reset_rx"), as shown in the following code snippet:

```
  for (;;)
```

```
{
  if(xQueueReceive(xQueueEnvio, szBufferTx,
    portMAX_DELAY) == pdPASS)
  {
    bPurgeRx = strcmp(szBufferTx, "cmd:reset_rx") == 0;

    if(bPurgeRx)
    {
      xQueueReset(xQueueRecepcao);
      continue;
    }
    // Treatment if it is packet to be sent or another command
  }
}
```

More C++

We use C ++ so far by calling only Arduino object methods (Serial basically). But what if we created, for example, a channel class, in the sense of abstracting the queue API for something like a data channel between tasks.

Here's an example of a header file to see what kind of thing you can do. We define a class that contains calls within the queue and shows an example of usage. This class is actually a class templaste (to be able to work with various data types in the queue) and is accompanied by operator overloads which allows a more interesting syntax with operators like << to add an item to >> to remove an item.

Note: After explaining the class we will present another listing using the .h file and instantiating channel objects to use the queues between tasks. The name `CanalEntreTasks` (portuguese) means: Channel Between Tasks.

Listing 10: Definition of the CanalEntreTasks class

```
#ifndef _CANALENTRETASKS_H_
#define _CANALENTRETASKS_H_

#include "funcoes_apoio.h"
/**
* Class that implements a channel, similar to those of golange encapsulating a queue
* CanalEntreTasks<T>
*/
template <typename T>

class CanalEntreTasks {
public:

   CanalEntreTasks(uint16_t pnsize):
     nTempoTimeoutMs(0), bResultadoOperacao(false)
   {
         xQueue = xQueueCreate(pnsize, sizeof(T));
   };

```

```
CanalEntreTasks(uint16_t pnsize, const char *pszNome):
  CanalEntreTasks(pnsize)
{
      setNome(pszNome);
};

~CanalEntreTasks() {
if(xQueue != NULL)
     vQueueDelete( xQueue );
};
//get Operation Result
bool getResultadoOperacao(void) const
{
  return bResultadoOperacao;
}

QueueHandle_t getFilaInterna()
{
  return xQueue;
}
void setNome(const char *pszNome) //set name
{
  strcpy(szNome, pszNome);
};

char * getNome(void) //get name
{
  return szNome;
};
// set Timeout value in ms
void setTempoTimeoutMs(const int pnTempoTimeoutMs)
{
  nTempoTimeoutMs = pnTempoTimeoutMs;
};
//send
void enviar(const T &pxValor, bool pbTimeOut = false)
{
  bResultadoOperacao = xQueueSendToBack( xQueue, &pxValor,
    pbTimeOut ? pdMS_TO_TICKS(nTempoTimeoutMs) :
portMAX_DELAY ) == pdPASS;
```

```cpp
   };
  //receive
   T receber(bool pbTimeOut = false) {
     T xValor;
      bResultadoOperacao = xQueueReceive( xQueue, &xValor,
        pbTimeOut ? pdMS_TO_TICKS(nTempoTimeoutMs) : portMAX_DELAY ) == pdPASS;

      return xValor;
   };
   //to see (peek)
   T ver() {
     T xValor;
      bResultadoOperacao = xQueuePeek( xQueue, &xValor, portMAX_DELAY ) == pdPASS;

      return xValor;
   };

//overwrite
void sobrescrever(T &pxValor) {
     T xValor;
      bResultadoOperacao = xQueueOverwrite( xQueue, &pxValor ) == pdPASS;

   };

private:
   QueueHandle_t xQueue;
   char szNome[33];
   int nTempoTimeoutMs;
   bool bResultadoOperacao;
};


// to allow chaining << operator (written without timeout, blocks while there is no space)
//Equivale a chamar CanalEntreTasks<T>::enviar(T xValor);
template <typename T>
CanalEntreTasks<T>& operator<<(CanalEntreTasks<T> &c, T pxValor)
```

```
{
     c.enviar(pxValor); //send
    return c;
}

// to allow the operator to be chained >> (read without timeout, blocks while not given)
// Equivalent to call T CanalEntreTasks <T>::receber();
template <typename T>
CanalEntreTasks<T>& operator>>(CanalEntreTasks<T> &c, T &pxValor)
{
    pxValor = c.receber(); //receive
    return c;
}

// to allow the operator to chain> to read (peek) the first item that would come out
// Equivalent to calling T ChannelEntreTasks<T>::ver();
template <typename T>
CanalEntreTasks<T>& operator>(CanalEntreTasks<T> &c, T &pxValor)

{
    pxValor = c.ver(); //to see (peek)
    return c;
}

// to allow the operator (override) to be chained to overwrite the last element
// It is equivalent to calling CanalEntreTasks<T>::sobrescrever(T &xValor);
template <typename T>
CanalEntreTasks<T>& operator<(CanalEntreTasks<T> &c, T &pxValor)

{
    c.sobrescrever(pxValor); //overwrite
    return c;
}
#endif

```

In line 14 we have the constructor of the class that receives by parameter the size of the class and in line 20 we have another option of constructor that besides the size, also receives a text for the name of the channel, useful for debugging the object differentiating it of others by calling the getNome () method defined from line 45.

See Appendix A.4 - Examples of Calls to Methods in the CanalEntreTasks Class where examples of calls are placed and also the link to github with actual use of this class.

Chapter 6 - Notification between tasks

Communication between tasks with notifications

So far we have used semaphores or queues to communicate events and data between tasks or between interrupts and taks. In these approaches we need to create objects and thus allocate more memory of one and another type of object with one or more positions (in the case of the queue).

We can, however, leverage the data structure already created for each task to send messages to it, notifying them at the bit level or at the value level.

In practice we use a function outside the task, which needs to know its identifier, to change the value associated with the notification by overwriting it, incrementing it or setting specific bits.

All this is done in an atomic way, protecting critical sections and what is more important, relatively simple for the programmer.

Within the task it is verified if a notification arrived, in a very similar way the reading of a row, but a little more complex.

Bit-level notification

In our example we will treat external interrupts 0 and 1 and send the value of a structure in the least significant part of the notification value for external interrupt 0 and in the most significant part for interrupt 0. Since the value is 32 bits, each structure can occupy up to 16 bits.

This structure contains the ID, the time in ticks between interrupts and a flag bit to make it easier to detect the event when reading the value, as can be seen below:

```
// purpose to map 16 bits
typedef union {
struct {
uint16_t nflagEventOccurred  : 1;
uint16_t nID                 : 4;
uint16_t nTimeBetweenStartUp : 11;
};
uint16_t nValue;
} button_read_t;
```

The bit mapping strategy was used to ensure that everything fits into 16 bits. The item corresponding to the time (nTimeBetweenStartUp) was impaired because it will only have a value of 11 bits, which would allow it to count intervals of a maximum of 30 seconds between the interrupts [5] , but in general this whole structure serves as an example of the possibility of passing complex data in the event.

Follow the code of the example with the two interrupts sending events to a task that changes the flashing time of the LED corresponding to the key according to the time between drives and also sends by the serial value read, so we can better follow what happens.

Listing 11: NotificacaoTasksDoisLEDs

```
#include <Arduino_FreeRTOS.h>
#include <task.h>

// Structure for interrupt reading (with 16 bits)
typedef union {
struct {
         uint16_t nflagEventOccurred     : 1;
         uint16_t nID                    : 4;
         uint16_t nTimeBetweenStartUp : 11;
       };
       uint16_t nValue;
} button_read_t;

// Bitmap with bit nflagEventImage set for use in masks
#define BUTTON_MASK_READING_16BIT_FLAGEVENT1_OCCURRED
((uint16_t)0x0001)

#define BUTTON_MASK_READING_32BIT_FLAGEVENT2_OCCURRED \
(((uint32_t)BUTTON_MASK_READING_16BIT_FLAGEVENT1_OCCURRED)<<16)

// Definition of LED pins
#define LED_BOARD LED_BUILTIN
#define LED_ADDITIONAL 12

static TaskHandle_t gxTaskNotification = NULL;

static void EXTI0_IRQHandler(void);
static void EXTI1_IRQHandler(void);

void setup()
{
  Serial.begin(9600);

  pinMode(LED_BOARD, OUTPUT);
  pinMode(LED_ADDITIONAL, OUTPUT);

  xTaskCreate(taskUserButtonFunc, "taskUserButtonFunc",
    128, NULL, 2, &gxTaskNotification );

  attachInterrupt(0, EXTI0_IRQHandler, RISING);
  attachInterrupt(1, EXTI1_IRQHandler, RISING);
}

void loop()
{
    //nothing to do
}

void EXTI0_IRQHandler(void)
{
  static TickType_t xTimeofLastInterruption = 0;
  TickType_t xNow;
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;
  uint32_t ulValueTaskNotification;
  button_read_t read;

  xNow = xTaskGetTickCountFromISR();

  read.nID = 0;

  read.nTimeBetweenStartUp = xNow - xTimeofLastInterruption;
  read.nflagEventOccurred = 1;

  xTimeofLastInterruption = xNow;

  if(gxTaskNotification)
  {
   ulValueTaskNotification = read.nValue;
   ulValueTaskNotification <<= 0;
   ulValueTaskNotification &= 0x0000FFFF;

   // Send a notification directly to the task indicated by gxTaskNotification
   xTaskNotifyFromISR( gxTaskNotification,
    ulValueTaskNotification, /* ulValue */
    eSetBits, /* eAction parameter. */
    &xHigherPriorityTaskWoken );
   }

   if( xHigherPriorityTaskWoken )
   {
    taskYIELD ();
    }
}

void EXTI1_IRQHandler(void)
{
  static TickType_t xTimeofLastInterruption = 0;
  TickType_t xNow;
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;
  uint32_ t ulValueTaskNotification;

  button_read_t read;
  xNow = xTaskGetTickCountFromISR();
  read.nID = 1;

  read.nTimeBetweenStartUp = xNow - xTimeofLastInterruption;

  read.nflagEventOccurred = 1;

  xTimeofLastInterruption = xNow;

  if(gxTaskNotification)
  {
    ulValueTaskNotification = read.nValue;
    ulValueTaskNotification <<= 16;
    ulValueTaskNotification &= 0xFFFF0000;

    // Send a notification directly to the task indicated by gxTaskNotification
    xTaskNotifyFromISR( gxTaskNotification,
      ulValueTaskNotification, /* ulValue */
      eSetBits, /* eAction parameter. */
      &xHigherPriorityTaskWoken );
  }

  if( xHigherPriorityTaskWoken )
  {
    taskYIELD ();
  }
}

void taskUserButtonFunc( void *pvParameters )
{
```

```
union {
  uint32_t ulReads;
  button_read_t read[2];
} xData;

char szMsgread[100];
BaseType_t xTempos[2] = {portMAX_DELAY, portMAX_DELAY};
int LEDs[2] = {LED_BOARD, LED_ADDITIONAL};
int contador = 0;

for (;;)
{
  // blinks LEDS, except if you do not have the time
  for(int i=0; i<2; i++)
  {
    if(xTempos[i] == portMAX_DELAY) continue;

    digitalWrite(LEDs[i], contador&0x01 ? HIGH : LOW);
    vTaskDelay( xTempos[i] );
  }

  contador++;

  // wait for notification with events
  xTaskNotifyWait(0, //ulBitsToClearOnEntry
    0xFFFFFFFF, //ulBitsToClearOnExit
    &xData.ulReads, //pulNotificationValue
    portMAX_DELAY);

  for(int i=0; i<2; i++)
  {
    if(!xData.read[i].nflagEventOccurred)
      continue;

    // set corresponding LED time
    xTempos[i] = xData.read[i].nTimeBetweenStartUp;

    sprintf(szMsgread, "Event ID %d, Time: %d ms",
xData.read[i].nID, xData.read[i].nTimeBetweenStartUp);


    Serial.println(szMsgread);
  }
164:   }
165: }
```

By now you are more familiar and so we will focus on sending the notification and on your return. The sending is very similar between the two interrupts, so we will show the most complex, the one of the external interrupt 1:

```
100:   if(gxTaskNotification)
101:   {
102:     ulValueTaskNotification = read.nValue;
103:     ulValueTaskNotification <<= 16;
104:     ulValueTaskNotification &= 0xFFFF0000;
105:
106:     // Send a notification directly to the task indicated by
gTask Notification
108:     xTaskNotifyFromISR( gxTaskNotification,
109:       ulValueTaskNotification, /* ulValue */
110:        eSetBits, /* eAction parameter. */
111:       &xHigherPriorityTaskWoken );
112:   }
```

At line 100 it is tested whether the target task identifier has actually been set.

The read variable, declared in line 90 (of filled in subsequent rows) is copied to the variable `ulValueTaskNotification` (although this copy could be avoided). It is then shifted 16 bits to the left (as these bits will be set in the notification).

At line 104 a mask is made only to reinforce this, since the offset to the left zeroes the bits in any way. It's a style of my this, although in a high performance program to execute unnecessary instructions.

At line 108 finally is called the function for notification xTaskNotifyFromISR (), which passes in its first parameter the target task identifier.

In line 109 the value is passed to the notification and in line 110, the operation that must be done with this value (`eSetBits`), which indicates that the bits 1 of the passed value must be set and the others kept as they are.

It is this operation that allows each interrupt to send values in its bit of value, 16 bits. This mechanism could be used for up to 32 different interrupt items, if each one just sets one of the bits. So it is possible to realize with western mechanism is versatile and powerful.

Now let's review how the receipt is:

```
145:    // wait for notification with events
146:    xTaskNotifyWait(0, //ulBitsToClearOnEntry
147:      0xFFFFFFFF, //ulBitsToClearOnExit
148:      &xData.ulReads, //pulNotificationValue
149:      portMAX_DELAY);
```

In the task, at line 122, a `union` is defined to be able to treat the two parts of the value as an array`xData.read[]` , very conveniently allowing to go through the two parts by testing the bit of the flag that indicates that the notification has written data in that part of the value. It is the flags `xData.read[0].nflagEventOccurred e xData.read[1].nflagEventOccurred;`

The call to read the notified value is called on line 146. Note that unlike other functions, in this case it is not possible to specify where the data is read since it will always be in the running task.

In the first parameter we could pass a value with the bits set to 1, indicating which bits would already be zeroed when entering the function. The value 0 is passed, indicating that no bits should be reset.

In the second parameter we indicate the bits that must be zeroed in the task notification value as soon as the function returns. In our case we do not want to leave anything there (and only in the copy of the original value for `xData.ulReads`). Therefore, the value `0xFFFFFFFF` is passed. So all bits are cleaned and new notifications can be saved successfully.

At line 148 is passed the value for the variable of the readings already mentioned, which will be inspected by the for of line 151, which traverses the two parts and tests (at line 153) each flag bit to know if the event of that button has arrived .

Byte-level notification

It is also possible to send an integer value overriding the value of the last unread notification and also incrementing the previous value, which is very interesting. You can also use this notification as a semaphore. All this can be done by passing different values to the third parameter (`eAction`):

```
eNoAction
eSetBits
eIncrement
eSetValueWithOverwrite
eSetValueWithoutOverwrite
```

And for the case of semaphore (binary) as of the different functions `xTaskNotifyGive()` and `ulTaskNotifyTake()`.

Overwriting and increasing the value of the notification

Let's expand the code of the previous listing by adding a second task of lower priority that will go every 5 seconds to check if there is any notification for it. This notification will be sent from the existing task every time an event of external interrupt 0 is received (by the notification mechanism that has already been implemented).

We will work with sending notifications using the `eAction` parameter with the `eSetValueWithOverwrite` value, to start with the counter notification 0 and the `eIncrement` value (at each event receipt) to allow incrementing the value of the notification, which will be read by the other task a every 5 seconds plus the task time that handles the external interrupts stay on hold and give chance to the second task of lower priority to execute.

Using a counter notification means that not all notifications need to be received, but rather that when you receive them, you can know how many events have occurred. Have you noticed that your smart tv (depending on the manufacturer, I imagine) sometimes does not respond to the remote control to decrease the volume you press several times the moment you press, but after some time? Well, it should be running something more priority than that, but the interrupt part caught the commands and may have increased or decreased a value pertaining to the volume,that was read at some point [6] .

The following will be the suggested changes to this implementation and the source will be made available on github at:
https://github.com/maxback/multitarefa_na_pratica/tree/master/NotificacaoTasks

Create a task with lower priority to receive the notifications, saving your handle to send the notifications:

```
xTaskCreate(taskCounterExt0Func, "gxTaskCounterExt0",
128, NULL, 1, & gxTaskCounterExt0 );
```

In the existing task function, send a first notification to the value 0 and then with each notification of ext0, it sends an increment:

```
if( gxTaskCounterExt0 )
{
```

```
xTaskNotify(gxTaskCounterExt0, 0,  eSetValueWithOverwrite);
}

//…

// if ext 0 event sends notification to another task
if( (i == 0) && (gxTaskCounterExt0) )
{
xTaskNotify(gxTaskCounterExt0, 0, eIncrement);
}
```

Now the receipt, within the loop of the new task, would check every 5 seconds if it has a notification, sending it through the serial. (complete function follows):

```
void taskCounterExt0Func( void *pvParameters )
{
uint32_t ulReads;
char szMsgCounter[100];

for (;;)
{
// Check if you have notification, without waiting
if( xTaskNotifyWait(0, 0, &ulReads, 0) == pdPASS)
{
// print counter
sprintf(szMsgCounter, "Counter: %d", ulReads);

SerialDebugWithSemaphore(szMsgCounter);

};

// wait 5 seconds
vTaskDelay(5000 / portTICK_PERIOD_MS);
}
}
```

One detail that we do not explain, and you will notice if you inspect the code attentively, is that we start using a semaphore and a function to send data through the serial, to avoid resource access conflicts already discussed inChapter 4 - Working with Semaphores .

Chapter 7 - Software Timings

Let's re-imagine the strategy of implementing the loop in the main function, which we used (and there are still situations to use it) when we did not have an operating system at our disposal. It was common to have global variables for loop control, which were timeouts that were decremented in timer interrupts and monitored in the main loop (or even in the interrupt itself), if it was too critical to wait for your test in the main loop.

Excerpts of codes that were scattered between the interrupt and the main loop, in addition to possible functions. If the embedded system started to grow we would have to add more variables by spreading this monitoring through several points in the source files.

FreeRTOS and RTOS generally provide mechanisms that simplify the task of executing a function periodically or only once after a timeout. Allowing abstract control of a global variable and critical section issues that could occur, causing corruption of system execution.

FreeRTOS turns on and off some of its resources at compile time. Since the timer in the background is a task with which we communicate with the API functions, we must verify that it defines that the active one has a value of 1 in the file FreeRTOSConfig.h which is in the lib folder corresponding to freeRTOS in the arduino structure.

```
#define configUSE_TIMERS 1
```

If you do not have this value you can change it, or even copy the file to the folder of your project and put the new value only in this copy, to set this only for the current project and not for all.

Periodic execution of a function

So far we have used traffic lights or queues to communicate events and data between tasks or

Let's see an example of a function that will run periodically, as configured without manipulating any variable, just running the desired code.

Unlike the function associated with task this does not have an infinite loop (and should not have) and does not need to reload any value associated with the execution control.

Listagem 12: BlinkELEDTimerFunc

```
#include <Arduino_FreeRTOS.h>
#include <timers.h>

void TaskAnalogRead( void *pvParameters );

// Timer function
void callbackBlinkLED(TimerHandle_t xTimer);

void setup() {
    // Timer identifier
   TimerHandle_t gxTimer;

   pinMode(LED_BUILTIN, OUTPUT);

    // initialize serial communication at 9600 bits per second:
   Serial.begin(9600);
   while (!Serial);

   xTaskCreate(TaskAnalogRead, (const portCHAR *) "AnalogRead",
     128, NULL, 1, NULL);

// create timer
gxTimer = xTimerCreate("TimerBlinkLED",
1000 / portTICK_PERIOD_MS, // call func. from 1 in 1s
pdTRUE,    // with autoreload (periodical)
0, // timer ID 0 indicating LED flashing index at start
callbackBlinkLED );

   if(gxTimer != NULL)
     xTimerStart ( gxTimer, 0);
}

void loop()
{
    //
}

void callbackBlinkLED(TimerHandle_t xTimer )
{
   // * Several blink patterns, based on text:
```

```
  *  L - Led on for a period of one second
  *  D - Led off for a period of one second
  *  When you reach the end of the last character, repeat the pattern.
  *  Four different patterns are defined.
  */
  // access for 1 second and off for 4 (2x), on for 1 second and off for 9 (2x),
  // on for 5 seconds and off for 5 and off for 1 048 and on 4 (2x)
  // (opposite of the first)
  char szPatterns[] = "LDDDDLDDDD_LDDDDDDDDDLDDDDDDDDD_LLLLLDDDDDLLLLLDDDDD_DLLLLDLLLL";

  uint32_t nIndex;
  // Read from the Timer ID the index value
  nIndex = ( uint32_t ) pvTimerGetTimerID( xTimer );

  // decide according to the character of the pattern
  char nivel = szPatterns[nIndex] == 'L' ? HIGH : LOW;

  // if it is '_' keep current
  if(szPatterns[nIndex] != '_')
  {
    digitalWrite(LED_BUILTIN, nivel);
  }

  nIndex++;
  if(szPatterns[nIndex] == '\0')
  {
      nIndex = 0;
  }

  // Save the index value in the Timer ID
  vTimerSetTimerID( xTimer, ( void * ) nIndex );
}


void TaskAnalogRead(void *pvParameters)
{
  (void) pvParameters;
  for (;;)
```

```
079:    {
080:      int sensorValue = analogRead(A0);
081:      Serial.println(sensorValue);
082:      vTaskDelay(1);   // 1 tick delay (15ms)
083:    }
084: }
```

On line 23, inside the setup function is called the `xTimerCreate` function to create a new timer. The first parameter only provides a readable name for the timer (since tasks can have multiple timers with the same function). In the second parameter a value in ticks is passed to determine the period in which the timer function will be called, in line 24. In our case, 1 second (`1000 / portTICK_PERIOD_MS`) was specified.

The third parameter indicates whether the function is going to be periodic or not. In our case we pass the value `pdTRUE` .

The next parameter, the timer ID is a 32-bit general purpose value that can be retrieved inside the function. Let's start with a value of 0, indicating that the first item in the LED blink pattern must be executed (we will see this in the timer function).

It remains to pass the timer function on line 27. The function returns the timer identifier on line 23, which is saved in gxTimer, to be used in the `xTimerStart()` function, which will start its operation, then on line 30.

In the `callbackBlinkLED()` timer function, on line 38 we have an array of characters with the characters 'L', 'D' and '_' which indicate the LED states at each function call. 'L' for on, 'D' for off and '_' to keep as is (in practice separates different patterns in the string).

The function starts by copying the timer ID to the index variable, calling the `pvTimerGetTimerID` function with the current timer identifier, received by parameter in the timer function. Remember that we passed the value 0 when creating the timer.

This value is then used as the basis for activating the LED (or deleting it) and is changed to the next index until it reaches the last one, when it returns to the value 0.

An important detail

In order for a second execution of the timer function to access an index different from the LED standard string, the value of the index variable must be stored in the timer ID to be read later because the function ends and the values of its variables are lost (unlike the function of a task, which is in an infinite loop). This is done at line 71, where the new index value is saved to the current timer, using the `vTimerSetTimerID()` function, passing the timer identifier and the value (with a type cast to `void *`, which is the expected type by the function).

Single execution of a function

We can create a single execution timer, or single shot, by simply passing the false `pdFALSE` value when creating the same:

```
gxOneShotTimer = xTimerCreate("OneShotTimer",
10000 / portTICK_PERIOD_MS,
pdFALSE,
0,
callbackOneShotTimer );
```

And then?

Well, then you wait for a moment that you want to trigger the timer, such as the push of a button and call the function `xTimerStart()` .

If you complete the set time (such as 10 seconds in the example) the defined function is called once and then no more. A second call to a `xTimerStart()` is required for a new timer. If you want to reload the timer already in progress, just call a `xTimerStart()` that the 10 seconds (for example) restart.

If, however, any event occurs and you want to abort the timing, just call the `xTimerStop()` function before the time runs out.

When should a book end?

This is a very difficult question. This is stopping here, although without contemplating all the freeRTOS (which is understandable). But it presents very useful elements for you to be able to make your real applications with very rich features, that you will have at your disposal like a Swiss Army Knife.

However only the use of each tool of this imaginary knife will improve your skills and give you experience on the best applications.

It is my suggestion to start using freeRTOS in new projects, seeking knowledge on the freeRTOS website (https://www.freertos.org/Documentation/RTOS_book.html):

Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

FreeRTOS V10.0.0 Reference Manual

Book companion source code

Do not forget to make your evaluation of the book and if you prefer, you can use the e-mail address max.back@google.com with the book subject (so that I can better identify the e-mail).

Appendix A - Additional sources

In this appendix we have complete listings that have not been fully addressed (or resulting from improvements in addressed listings) and which may be copied depending on the reader being used. However, they are included here to facilitate a better examination.

Apendix A.1 - AnalogRead_DigitalREadModificadoVariosLeitores (Com suspensão da task)

```
   #include <Arduino_FreeRTOS.h>
    #include <semphr.h>  // add the FreeRTOS functions for
Semaphores (or Flags).

   // type of parameters
   typedef struct {
     const char *pcTaskName; // name of the task TO BE CREATED
     const char *pcText; // text to be sent by the serial before
reading, identifying its source
     int SensorID; // Analog input ID used
     int DelayTicks; // time between sending of readings, in ticks
   } AnalogReadParam_t;

   // Declare a mutex Semaphore Handle which we will use to manage
the Serial Port.
   // It will be used to ensure only only one Task is accessing
this resource at any time.
   SemaphoreHandle_t xSerialSemaphore;

   // define two Tasks for DigitalRead & AnalogRead
   void TaskDigitalRead( void *pvParameters );
   // accepts as parameter AnalogReadParam_t
   void TaskAnalogReadParam( void *pvParameters );

   // define parameters for three analog read tasks that accept
structure parameters
   AnalogReadParam_t xParams[3], *pxParam = xParams;

   // the setup function runs once when you press reset or power
the board
   void setup() {

     // initialize serial communication at 9600 bits per second:
    Serial.begin(9600);

    while (!Serial) {
      ; // wait for serial port to connect. Needed for native USB,
on LEONARDO, MICRO, YUN, and other 32u4 based boards.
    }

     // Semaphores are useful to stop a Task proceeding, where it
should be paused to wait,
     // because it is sharing a resource, such as the Serial port.
     // Semaphores should only be used whilst the scheduler is
running, but we can set it up here.
      if ( xSerialSemaphore == NULL )  // Check to confirm that the
Serial Semaphore has not already been created.
     {
       xSerialSemaphore = xSemaphoreCreateMutex();  // Create a
mutex semaphore we will use to manage the Serial Port
      if ( ( xSerialSemaphore ) != NULL )
        xSemaphoreGive( ( xSerialSemaphore ) );  // Make the
Serial Port available for use, by "Giving" the Semaphore.
     }

      // define parameters for three analog read tasks that accept
structure parameters

      pxParam->pcTaskName = "TaskInputA1";
      pxParam->pcText = "Input A1";
      pxParam->SensorID = A1;
      pxParam->DelayTicks = pdMS_TO_TICKS( 250UL );
      pxParam++;

      pxParam->pcTaskName = "TaskInputA2";
      pxParam->pcText = "Input A2";
      pxParam->SensorID = A2;
      pxParam->DelayTicks = 1; //pdMS_TO_TICKS( 250UL );
      pxParam++;

      pxParam->pcTaskName = "TaskInputA3";
      pxParam->pcText = "Input A3";
      pxParam->SensorID = A3;
      pxParam->DelayTicks = 1; //pdMS_TO_TICKS( 250UL );
      pxParam++;


       //xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)&xParams[0], 1  /* Priority  */, NULL );
      //xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128, (
void *)&xParams[1], 1  /* Priority  */, NULL );
      //xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)&xParams[2], 1  /* Priority  */, NULL );

       //armazena a última task criada
      TaskHandle_t xLastTask;
       //cria as tasks percorre array de ponteiros até o nulo (para
facilitar)
      for(pxParam = &xParams[0]; pxParam < &xParams[3]; pxParam++)
      {
```

```
      {
         Serial.println("\n---------------------------------");
         Serial.print("Creating ");
         Serial.println(pxParam->pcTaskName);

         xTaskCreate(TaskAnalogReadParam, pxParam->pcTaskName, 128,
(void *)pxParam, (configMAX_PRIORITIES - 2)  /* Priority  */ ,
&xLastTask );
      }

      vTaskSuspend(xLastTask);

        // Now set up two Tasks to run independently.
      xTaskCreate(
         TaskDigitalRead
         ,  (const portCHAR *)"DigitalRead"   // A name just for
humans
         ,  128  // This stack size can be checked & adjusted by
reading the Stack Highwater
         ,  (void *)xLastTask //Passa o valor do hadle da task com
cast para (void *)
         ,  (configMAX_PRIORITIES - 3)  // Priority, with 3
(configMAX_PRIORITIES - 1) being the highest, and 0 being the
lowest.
         ,  NULL );

   // Now the Task scheduler, which takes over control of
scheduling individual Tasks, is automatically started.
   }

   void loop()
   {
      // Empty. Things are done in Tasks.
   }

   void SerialDebugWithSemaphore(const char *pszText)
   {
      if ( xSemaphoreTake( xSerialSemaphore, ( TickType_ t ) 5 ) ==
pdTRUE )
      {
         Serial.print(pszText);

         xSemaphoreGive( xSerialSemaphore ); // Now free or "Give"
the Serial Port for others.
      }
   }

   /*--------------------------------------------------*/
   /*---------------------- Tasks ---------------------*/
   /*--------------------------------------------------*/

    void TaskDigitalRead( void *pvParameters)  // This is a Task.
   {
      /*
      DigitalReadSerial
      Reads a digital input on pin 2, prints the result to the
serial monitor

      This example code is in the public domain.
     */

     // digital pin 2 has a pushbutton attached to it. Give it a
name:
     uint8_t pushButton = 2;

     TaskHandle_t xTaskParaReiniciar = (TaskHandle_t) pvParameters;

      // make the pushbutton's pin an input:
     pinMode(pushButton, INPUT);

      for (;;) // A Task shall never return or exit.
     {
       // read the input pin:
      int buttonState = digitalRead(pushButton);

       // See if we can obtain or "Take" the Serial Semaphore.
      // If the semaphore is not available, wait 5 ticks of the
Scheduler to see if it becomes free.
      if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) ==
pdTRUE )
      {
         //Envia o Text recebido por parâmetro
         Serial.print("digital input:") ;

         // We were able to obtain or "Take" the semaphore and can
now access the shared resource.
         // We want to have the Serial Port for us alone, as it
takes some time to print,
         // so we don't want it getting stolen during the middle of
a conversion.
         // print out the state of the button:
         Serial.println(buttonState);

         xSemaphoreGive( xSerialSemaphore ); // Now free or "Give"
the Serial Port for others.

         //Decide se reinicia a task
         if( (xTaskParaReiniciar != NULL) && (buttonState == 1) )
```

```
        {
         vTaskResume(xTaskParaReiniciar);
        }

      }

       vTaskDelay(1);  // one tick delay (15ms) in between reads
for stability
     }
    }


    void TaskAnalogReadParam( void *pvParameters )  // This is a
Task.
    {
      //pegando o parametro como o Text da string
     AnalogReadParam_t *pxParams = (AnalogReadParam_t *)
pvParameters;

     for (;;)
     {
       // read the input on analog pin:
      int sensorValue = analogRead(pxParams->SensorID);

       // See if we can obtain or "Take" the Serial Semaphore.
      // If the semaphore is not available, wait 5 ticks of the
Scheduler to see if it becomes free.
      if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) ==
pdTRUE )
      {
        //Envia o Text recebido por parâmetro
        Serial.print(pxParams->pcText);
        Serial.print(": ");
        // We were able to obtain or "Take" the semaphore and can
now access the shared resource.
        // We want to have the Serial Port for us alone, as it
takes some time to print,
        // so we don't want it getting stolen during the middle of
a conversion.
        // print out the value you read:
        Serial.println(sensorValue);

        xSemaphoreGive( xSerialSemaphore ); // Now free or "Give"
the Serial Port for others.

      }

       vTaskDelay(pxParams->DelayTicks);  // one tick delay (15ms)
```

```
in between reads for stability
    }
   }
```

Appendix A.2 - Exercise proposed in Chapter 5

```
#include <Arduino_FreeRTOS.h>
#include <queue.h>

const byte KeyPin = 2;
// New queue to receive key bindings
QueueHandle_t xQueueEventosChave;
// Queue identifier
QueueHandle_t xQueue;

// Function of new task to receive chave events
void TaskEventosChave( void *pvParameters );
// Task to notify the key bind
void TaskSend( void *pvParameters );

void setup() {
 BaseType_t xRet;

 Serial.begin(9600);
 attachInterrupt(digitalPinToInterrupt(KeyPin),
  tratarInterrupcaoKeyPin, RISING);
 pinMode(KeyPin, INPUT_PULLUP);

  // Create the event queue of the key with TICKS COUNTER ITEM
 xQueueEventosChave = xQueueCreate(
   10,                  // Queue size: 10
   sizeof( TickType_t )    // Size of the item: Size of the type
 );

 if(xQueueEventosChave) Serial.println("Fila xQueueEventosChave
criada com sucesso!");

  // Creates the character submission queue
 xQueue = xQueueCreate(
   10,                  // Queue size: 10
   21 *sizeof( char )   //  Size of the item: 21
 );

 if(xQueue) Serial.println("Fila criado com sucesso!");

 if(xTaskCreate(TaskEventosChave, (const portCHAR *)
"TaskEventosChave",  128, NULL, 2, NULL) == pdPASS)
 {
   Serial.println("Task TaskEventosChave criada com sucesso:");
 }

 if(xRet == pdPASS) Serial.println("Task criada com sucesso:");

 xRet = xTaskCreate(TaskSend, (const portCHAR *) "TaskSend",  128,
NULL, 2, NULL);
 if(xRet == pdPASS) Serial.println("Task criada com sucesso:");
}

void loop() {
  //Monitoramento da flag saiu daqui
}

void tratarInterrupcaoKeyPin() {
 BaseType_t xHigherPriorityTaskWoken;
 TickType_t xContadorTicks;
 xContadorTicks = xTaskGetTickCountFromISR();
  //coloca o valor de ticks atual na fila
  xQueueSendToBackFromISR( xQueue, &xContadorTicks,
&xHigherPriorityTaskWoken);
 if( xHigherPriorityTaskWoken )
 {
  taskYIELD ();
 }
}

//Função da task para tratar evento da chave
void TaskEventosChave( void *pvParameters ) {
 TickType_t xTimeofLastInterrupt;
 TickType_t xNow;
 int nTempoEmMs;
  //Armazenada o Text que será recebido
 char szText[21];
 Serial.println("Starting loop: TaskEventosChave");
  //coloca o contador no inicio como se fosse da primeira int.
 xTimeofLastInterrupt = xTaskGetTickCount();
  //loop da task
 for( ;; )
 {
   //Espera indefinidamente até ler um item da fila
  xQueueReceive( xQueueEventosChave, &xNow, portMAX_DELAY );
  nTempoEmMs = (xNow - xTimeofLastInterrupt) * portTICK_PERIOD_MS;
  sprintf(szText, "Botão: %d ms", nTempoEmMs);
  xQueueSendToBackFromISR( xQueue, szText, portMAX_DELAY);

  xTimeofLastInterrupt = xNow;
 }
}

//Função da task para notificar o ligamento da chave
void TaskSend( void *pvParameters ) {
  //Armazenada o Text que será recebido
 char szText[21];

 Serial.println("Starting loop: TaskSend");

  //loop da task
 for( ;; )
 {
   //Espera indefinidamente até ler um item da fila
  xQueueReceive( xQueue, &szText, portMAX_DELAY );

  Serial.print(szText);
 }
}
```

Appendix A.3 - Compact 9 source for copying to the IDE

```
#include <Arduino_FreeRTOS.h>
#include <queue.h>

typedef enum {itTemperatureReading, itHourry, itReceivedText}
ItemType_t;
//Type para os três Types de Data (usa union) para interpretar a
memÓria de cada jeito
typedef struct {
 ItemType_t Type;
 union {
  float Temperature;
  struct {
   uint8_t Hour;
   uint8_t Minutes;
   uint8_t Seconds;
  } Horary;
  char Text[13];
 } Data;
} QueueItem_t;

//Identificador da fila unica
QueueHandle_t xQueue;

void TaskReadTemp( void *pvParameters );
void TaskCurrentTime( void *pvParameters );
void TaskSerialReception( void *pvParameters );
void TaskSend( void *pvParameters );

int createTasks(void)
{
  if(xTaskCreate(TaskReadTemp, (const portCHAR *)
"TaskReadTemp",  128, NULL, 2, NULL) == pdPASS)
   Serial.println("Task TaskReadTemp criada com sucesso!");
  else
   return 0;

  if(xTaskCreate(TaskCurrentTime, (const portCHAR *)
"TaskCurrentTime",  128, NULL, 2, NULL) == pdPASS)
   Serial.println("Task TaskCurrentTime criada com sucesso!");
  else
   return 0;

  if(xTaskCreate(TaskSerialReception, (const portCHAR *)
"TaskSerialReception",  128, NULL, 2, NULL) == pdPASS)
   Serial.println("Task TaskSerialReception criada com sucesso!");
  else
   return 0;

  if(xTaskCreate(TaskSend, (const portCHAR *) "TaskSend",  128,
NULL, 2, NULL) == pdPASS)
   Serial.println("Task TaskSend criada com sucesso!");
  else
   return 0;

  return 1;
}

void setup() {
 BaseType_t xRet;
 Serial.begin(9600);

 xQueue = xQueueCreate(10, sizeof( QueueItem_t ));
 if(xQueue)
 {
  Serial.println("Fila criado com sucesso!");
  if(!createTasks())
   Serial.println("Ocorreu Error creating as tasks!");
 }
 else
  Serial.println("Ocorreu Error creating a fila!");
}

void loop() {
}

void TaskReadTemp( void *pvParameters )
{
 QueueItem_t Item;
 Item.Type = itTemperatureReading;

 Serial.println("Starting loop: TaskReadTemp");
 for( ;; )
 {
  int sensorValue = analogRead(A0);
  Item.Data.Temperature = (float(sensorValue)*5/(1023))/0.01;
  xQueueSendToBack(xQueue, &Item, portMAX_DELAY);
  vTaskDelay(500 / portTICK_PERIOD_MS);
 }
}

void treatClock( QueueItem_t *pItem)
{
  pItem->Data.Horary.Seconds++;
  if(pItem->Data.Horary.Seconds == 60)
  {
   pItem->Data.Horary.Seconds = 0;
   pItem->Data.Horary.Seconds = 0;
   pItem->Data.Horary.Minutes++;

   if(pItem->Data.Horary.Minutes == 60)
   {
    pItem->Data.Horary.Minutes = 0;
    pItem->Data.Horary.Hour++;

    if(pItem->Data.Horary.Hour == 24)
     pItem->Data.Horary.Hour = 0;
   }
  }
}

void TaskCurrentTime( void *pvParameters )
{
 QueueItem_t Item;
 Item.Type = itHourry;
 Item.Data.Horary.Hour = 0;
 Item.Data.Horary.Minutes = 0;
 Item.Data.Horary.Seconds = 0;

 Serial.println("Starting loop: TaskSerialReception");
 for( ;; )
 {
  treatClock(&Item);
  xQueueSendToBack(xQueue, &Item, portMAX_DELAY);
  vTaskDelay(1000 / portTICK_PERIOD_MS);
 }
}

void TaskSerialReception( void *pvParameters )
{
 QueueItem_t Item;
 char readedChar;
 int size = 0 ;

 Item.Type = itReceivedText;
 Item.Data.Text[size] = '\0';

 Serial.println("Starting loop: TaskSerialReception");
 for( ;; )
 {
  if(Serial.available())
  {
   readedChar = (char)Serial.read();
   if( readedChar != '\n')
   {
    Item.Data.Text[size++] = readedChar;
    Item.Data.Text[size] = '\0' ;
   }
   if( (readedChar == '\n') || (size == (sizeof(Item.Data.Text)-1))
)
   {
     xQueueSendToBack(xQueue, &Item, portMAX_DELAY);
     size = 0;
     Item.Data.Text[size] = '\0';
   }
  }
 }
}

void TaskSend( void *pvParameters ) {
 char DisplayMirror[2][17] = {"00:00:00 |      ",
                     "T: 000.0 |      "};
 QueueItem_t Item;
 int l,i;
 char *p;

 Serial.println("Starting loop: TaskSend");

 for( ;; )
 {
  xQueueReceive( xQueue, &Item, portMAX_DELAY );
  switch(Item.Type)
  {
   case itTemperatureReading:
    sprintf(&DisplayMirror[1][3], "%3.1f", Item.Data.Temperature);
   break;
   case itHourry:
    sprintf(&DisplayMirror[0][0], "%0.2d" ,
     Item.Data.Horary.Hour);
    sprintf(&DisplayMirror[0][3], "%0.2d",
     Item.Data.Horary.Minutes);
    sprintf(&DisplayMirror[0][6], "%0.2d",
     Item.Data.Horary.Seconds);
   break;
   case itReceivedText  :
    for(l=0; l<2; l++)
    {
     for(i=0, p = &DisplayMirror[l][10]; i<6; i++, p++) *p = ' ';
     *p = '\0';
    }

    for(l=0; l<2; l++)
    {
     p = &DisplayMirror[l][10];
     for(i=0; i<6 && Item.Data.Text[i]; i++)
     {
        *p = Item.Data.Text[i];
        p++;
     }
     *p = '\0';
    }
   break;
  }
  Serial.println("----------------");
  Serial.println(DisplayMirror[0]);
  Serial.println(DisplayMirror[1]);
 }
}
```

Appendix A.4 - Examples of Calls to Methods in the CanalEntreTasks Class

In this appendix we have "loose" snippets of uses of the CanalEntreTasks class and in github you can see these codes in group, treateds to communicate between tasks with opium flares called pipelines, that is, input and output queues on the nodes and tasks using these structures to read from a channel, process the data and put it on another channel.

This code is on the link https://github.com/maxback/multitarefa_na_pratica/tree/master/CanaisSemelhant 7 , if you want to walk around it or even download it and see its finalization in action.

Mas seguem os trechos de código:

```
#include "CanalEntreTasks.h"

//Type de parameters
typedef struct {
 const char *pcTaskName;
 const char *pcText;
 int SensorID;
 int DelayTicks;
 QueueHandle_t xQueueCalculo;
 CanalEntreTasks<int> *CanalEnvio;
} AnalogReadParam_t;

CanalEntreTasks<int> *CanalEnvio;

...
     if(pxParams->CanalEnvio != NULL)
     {
      //pxParams->CanalEnvio->enviar(sensorValue);

      //Teste com o operador de override:
      /*
      //envia para o canal
      *(pxParams->CanalEnvio) << 123;
      //e sobrescreve em seguida
      *(pxParams->CanalEnvio) < sensorValue;
      * /
```

```
          *(pxParams->CanalEnvio) << sensorValue;

          //poderiam ser varios valores
          //*(pxParams->CanalEnvio) << sensorValue-1 << sensorValue
<< sensorValue+1;

          /*
          (*pxParams->CanalEnvio) << 123;

          operator<<(   operator<<((*pxParams->CanalEnvio), 99),
sensorValue) ;

          operator<<((*pxParams->CanalEnvio),
99).enviar(sensorValue);
          operator<<((*pxParams->CanalEnvio),
sensorValue).enviar(123);


          (*pxParams->CanalEnvio) << sensorValue << 123;
          */
        }
     }


   typedef struct {
    CanalEntreTasks<int> *entrada;
    CanalEntreTasks<int> *saida;
   } CanaisDaTask_t;

   // Define 3 canais e 3 funcoes de tasks
   CanalEntreTasks<int> xCanalMultiplica(10, "CanalEntMultiplica");
   //CanalEntreTasks<int> xCanalDiminui(10);
   //CanalEntreTasks<int> xCanalImprime(30);

   CanaisDaTask_t C1 = {&xCanalMultiplica, new CanalEntreTasks<int>
(10, "CanalEntDiminui")};
   CanaisDaTask_t C2 = {C1.saida, new CanalEntreTasks<int>(30,
"CanalImprime")};
   CanaisDaTask_t C3 = {C2.saida, NULL};

      //valor = C->entrada->receber();
      *(C->entrada) >> valor ;

      SerialDebugWithSemaphore("\nTaskMultiplica recebeu ", valor);
       *(C->saida) << valor;
      *(C->entrada) >> valor;

         //C->saida->enviar(diminuido);
       *(C->saida) << valor;

         SerialDebugWithSemaphore("\n   -> DEPOIS TaskDiminui
enviando para", C->saida->getNome(), ": ",  valor);
```

```
//seta timeout desta fila
C->entrada->setTempoTimeoutMs(100);

 *(C->entrada) >> valor;

 //*(C->entrada) >> valor;

    *(C->saida) << valor;

    *(pipeline_getCanalEntrada()) << valor;
```

Notas

[←1]

In the implementation of the FreeRTOS port being worked on in this book the loop () function has been associated with the RTOS IDLE function, that is, it is called from time to time. This is an advanced topic, so we can assume by the hour, that this function should not receive any code.

[←2]

But it may happen that the second tasks will be removing items while the first one is adding items. Example: Before the item with value 3 is queued the item with value 1 has already been withdrawn.

[←3]
As commented before, an out-of-memory error might occur. This type of situation arises when we start making larger systems and the available memory for dynamic allocation is insufficient.

[←4]

One possible solution to this is to use a freeRTOS software timing for a timeout control.

[←5]

Since each tick corresponds to 15ms and with 11 bits we get values from 0 to $2^{11}-1$ (2047), the maximum value of the interval will be 2047 x 15ms = 30705ms or approximately 30 seconds

[←6]

Of course this is an example I'm imagining. It is also possible that the implementation has been in a queue, or some other implementation strategy. It is always important to evaluate the cost to the system and the priority of one task over another, thus extracting the best possible performance for critical activities and leaving the others with acceptable performance.

[←7]
The names and comments in this example are in Portuguese.

Table of Contents