Practical
ESP32 Multitasking (4)
Binary semaphores

By **Warren Gay** (Canada)

When multitasking in FreeRTOS, there is often a need to synchronize between tasks. One such synchronization facility is the binary semaphore. This article explores the binary semaphore and illustrates it with a simple demonstration.

What is a semaphore?

A semaphore is a function that permits the caller to block its continued execution until permission is "given" to proceed. In other words, the caller tries to "take" the semaphore but, if the semaphore is already taken, the calling task waits (blocks). This is much like boys wishing to select a popular girl at a dance. If the girl is currently dancing (taken), then the other interested boys must wait until she is ready to "give" herself again. This leads us to define one of the properties of a semaphore: a *binary* semaphore has two states:

- taken
- given

How does the semaphore protect?

A semaphore itself cannot control access to any resource. It is only by *agreement* that resource protection may be implemented via a semaphore. If the boys at the dance didn't respect the concept of "taken" and "given", then multiple boys could grab the girl by the arms in a struggle. The semaphore operates by protocol: the accessor agrees to use any resource on offer only if it succeeds in taking the semaphore. It will also not use the resource again until it has taken the next available semaphore.

Timeouts when taking

A boy waiting at the dance may have limited patience. He may decide that, after ten minutes of waiting, he will try for a different girl to dance with. Binary semaphores also allow the caller to specify such a timeout period in system ticks. If an attempt to take a semaphore takes longer than the specified number of ticks, the call will return with a fail code. Alternatively, FreeRTOS can be told to wait indefinitely until the semaphore has been given (much like a boy that is smitten). Finally, there is the option to specify no timeout at all with the attempt failing immediately if the semaphore cannot be taken.

To summarize, semaphores can operate with time in three ways:

- Block forever, until the semaphore can be "taken" by the caller.
- Block for a defined period of time, failing if the operation times out.
- Fail immediately if the semaphore is currently "taken".

Ownership and giving

When a semaphore is taken, the task is said to "own" it. The boy who is currently dancing with the girl then effectively "owns" her (but perhaps not her heart). When he has finished dancing with her, he can "give" her to someone else immediately or simply give her hands back, freeing her: there is no waiting involved. Likewise, when "giving" a semaphore, there is no need for a timeout argument. The giving of an owned semaphore happens immediately.

The act of "giving" a semaphore does not *necessarily* imply that it will be immediately "taken" by another task. Once the boy has finished dancing with the girl she becomes available, but that doesn't mean that there are necessarily other interested takers. The other boys may have given up and found different dance partners.

Initial state

Let's strain this analogy a little further – the girl is created in the "taken" state by her parents. Only when her parents allow her to attend the dance is she "given" and becomes available. In the same way, the FreeRTOS binary semaphore is initially created in the "taken" state. This differs from mutexes in Linux or Windows that you may be comparing it to, as they are created in a "given" state. More will be said about the FreeRTOS mutex in a later installment.

xSemaphoreBinaryCreate():

To create a binary semaphore in FreeRTOS is simple:

```
SemaphoreHandle_t h;

h = xSemaphoreCreateBinary();
assert(h)
```

There are no arguments to supply, and a handle to the binary semaphore that was created is returned. It is possible that the handle will be returned as `nullptr` (`NULL`) if you've exhausted the available memory. It is recommended to check the returned value (the `assert()`

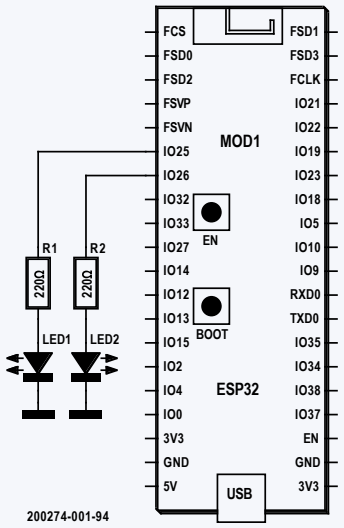

Figure 1. Schematic for the semaphores.ino demonstration.

macro from assert.h was used here) to ensure the semaphore was created.

xSemaphoreGive()

To give a semaphore is also straight forward:

```
SemaphoreHandle_t hMySemaphore;
BaseType_t rc; // return code

...
rc = xSemaphoreGive(hMySemaphore);
assert(rc == pdPASS);
```

The "give" operation can fail, returning `pdFAIL`, only for the following reasons:

- The handle (`hMySemaphore`) is invalid.
- The semaphore has already been "given".

xSemaphoreTake()

Taking a semaphore is potentially a blocking operation for the calling task:

LISTING 1: THE SEMAPHORES.INO DEMONSTRATION PROGRAM [1].

```
0001: // semaphores.ino
0002: // Practical ESP32 Multitasking
0003: // Binary Semaphores
0004:
0005: #define LED1_GPIO   25
0006: #define LED2_GPIO   26
0007:
0008: static SemaphoreHandle_t hsem;
0009:
0010: void led_task(void *argp) {
0011:   int led = (int)argp;
0012:   BaseType_t rc;
0013:
0014:   pinMode(led,OUTPUT);
0015:   digitalWrite(led,0);
0016:
0017:   for (;;) {
0018:     // First gain control of hsem
0019:     rc = xSemaphoreTake(hsem,portMAX_DELAY);
0020:     assert(rc == pdPASS);
0021:
0022:     for ( int x=0; x<6; ++x ) {
0023:       digitalWrite(led,digitalRead(led)^1);
0024:       delay(500);
0025:     }
0026:
0027:     rc = xSemaphoreGive(hsem);
0028:     assert(rc == pdPASS);
0029:   }
0030: }
0031:
0032: void setup() {
0033:   int app_cpu = xPortGetCoreID();
0034:   BaseType_t rc;  // Return code
0035:
0036:   hsem = xSemaphoreCreateBinary();
0037:   assert(hsem);
0038:
0039:   rc = xTaskCreatePinnedToCore(
0040:     led_task,   // Function
0041:     "led1task", // Task name
0042:     3000,       // Stack size
0043:     (void*)LED1_GPIO, // arg
0044:     1,          // Priority
0045:     nullptr,    // No handle returned
0046:     app_cpu);   // CPU
0047:   assert(rc == pdPASS);
0048:
0049:   // Allow led1task to start first
0050:   rc = xSemaphoreGive(hsem);
0051:   assert(rc == pdPASS);
0052:
0053:   rc = xTaskCreatePinnedToCore(
0054:     led_task,   // Function
0055:     "led2task", // Task name
0056:     3000,       // Stack size
0057:     (void*)LED2_GPIO, // argument
0058:     1,          // Priority
0059:     nullptr,    // No handle returned
0060:     app_cpu);   // CPU
0061:   assert(rc == pdPASS);
0062: }
0063:
0064: // Not used
0065: void loop() {
0066:   vTaskDelete(nullptr);
0067: }
```

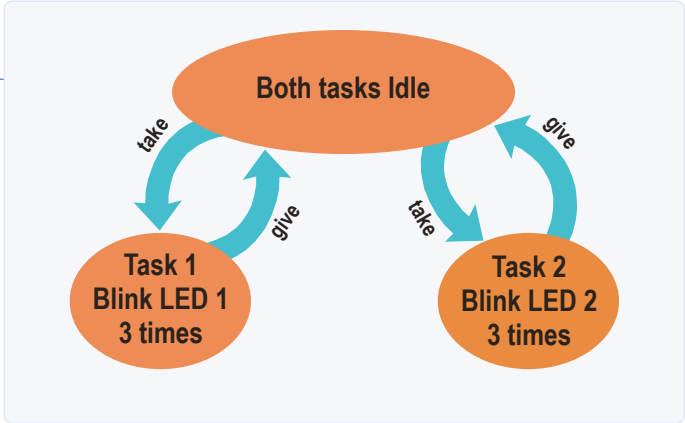

Figure 2. The states of the executing tasks in the semaphores.ino program.

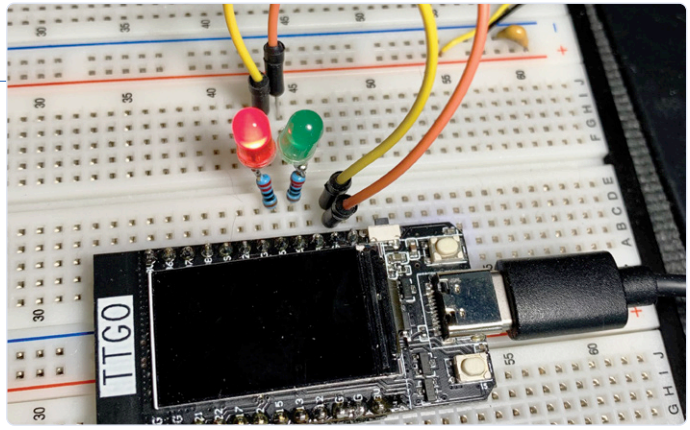

Figure 3. The ESP32 TTGO driving two LEDs using the program semaphores.ino.

```
SemaphoreHandle_t hMySemaphore;
TickType_t wait = 30; // ticks
BaseType_t rc; // return code

...
rc = xSemaphoreTake(hMySemaphore, wait);
// Returns pdPASS when taken,
// otherwise returns pdFAIL if it times out
```

The "take" operation can fail if the handle is invalid or the call has timed out (period as defined by the `wait` argument). Otherwise, the call will block the calling task's execution until it has "taken" the semaphore. The `wait` parameter can have one of three different values according to the caller's requirements:

- Macro value `portMAX_DELAY` – wait forever until "taken".
- Some positive non-zero value – wait for so many ticks.
- Zero – fail immediately if the semaphore cannot be "taken" immediately.

Demonstration

As a demonstration, two tasks will each blink their own LED (see **Listing 1**). By sharing a binary semaphore, only one of the tasks will be allowed to blink their LED at any one time. The non-running task must wait until the semaphore has been "given" by the opposing running task, making the semaphore available to be "taken" once more. **Figure 1** illustrates the schematic for the LED hookup for any ESP32 you choose to use.

Figure 2 illustrates the states of the pair of tasks executing in the demo program. When one task takes the semaphore, that owning task is able to execute and blink its LED, while the other is blocked waiting. Only when the owning task gives the semaphore back can the other task take it and execute. **Figure 3** illustrates an example breadboard setup using the TTGO ESP32 dev board.

In the `setup()` routine, line 36 creates the semaphore and assigns the handle to the static global variable `hsem`. Lines 39 to 46 creates the first task to blink LED1 (note the argument in line 43 of the task creation call). The GPIO pin to be used is passed as a `void` pointer. This is then cast back to an `int` GPIO value in the task function `led_task()` (line 11), of the task function. This is abusing the pointer to pass a value, but this works as long as the value fits within the same number of bits as a pointer address.

After the first task is created, we "give" the semaphore in line 50. By doing this before the creation of the second task makes it likely that the first task acquires the semaphore first. Lines 53 to 60 then creates the second task. Both tasks run the same function code `led_task()` but with different argument values specifying the LED GPIO pin to be used. Lines 14 and 15 configure the LED being used within the task. The task then enters an endless loop starting at line 17. The first step performed within this loop is to "take" the semaphore (line 19). Only one task at a time will succeed at this.

The task that succeeds in taking the semaphore will continue through the loop in lines 22 to 25. This loop blinks the task's LED three times. Once complete, the semaphore is "given" in line 27, allowing the other task to "take" the semaphore. In this way, each task takes turns in handing control over to the other task.

Summary

The binary semaphore used in this demonstration was exercised in two different ways. Before the first "give" operation in the `setup()` function, neither task is able to proceed. This caused the semaphore to behave as a barrier because no task was able to proceed. After that first "give" operation, one of the two tasks "takes" that semaphore, blinks their LED, and then "gives" the semaphore back. Operating in this manner, the semaphore seems to behave like a mutex. Both tasks continually attempt to execute their code but, through the mutual exclusion of the semaphore, only one task is ever permitted to blink its LED.

It is important to memorize that the binary semaphores are created in the "taken" state (unlike a mutex). If the semaphore in this demonstration had not initially been given in the `setup()` routine (line 50), both tasks would have been stuck waiting forever. Other synchronization primitives are available within FreeRTOS (including the mutex), but the simple binary semaphore is sometimes all that is needed.

200274-01

WEB LINK

[1] https://github.com/ve3wwg/esp32_freertos/blob/master/semaphores/semaphores.ino