Practical ESP32 Multitasking (2)

Task priorities

By **Warren Gay** (Canada)

In microcontroller projects, developers often face the problem that many processor tasks need to be performed at a time. ESP32 and Arduino IDE make task programming easy, as the popular FreeRTOS is already integrated into the core libraries [1]. In this second part of the series we are especially dealing with task priorities.

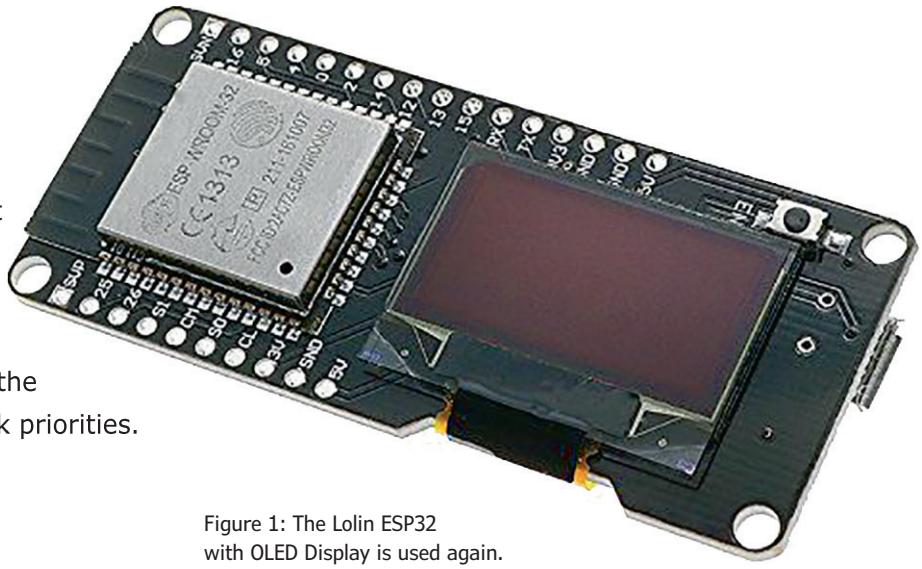

Figure 1: The Lolin ESP32 with OLED Display is used again.

Within the ESP32 implementation of the FreeRTOS scheduler, tasks are executed according to their priority. Priority is assigned when the task is created but can be altered later. Higher-numbered tasks are considered first for the configured CPU, while zero-priority tasks are considered last. Execution priority may be a familiar concept but the FreeRTOS real-time scheduler works differently than what you might be used to with Linux or Windows. This article will explore the difference using a demonstration.

The ESP32 implementation includes a maximum of 25 priority levels, ranging from zero to 24. By default, the Arduino `setup()` and `loop()` functions run at priority level 1 (recall that these functions are from the same main task [1]).

Vive la différence

How different can task scheduling be? On a Linux system for example, priority affects the relative urgency of the process or thread. Even a low priority process gets some CPU time — normally taking longer to run. But the process **does** always execute eventually. This is where the difference lies.

In a real-time system like FreeRTOS, the scheduler does **not** guarantee that lower-priority tasks will ever get executed. For example, if you have priority-9 tasks that are always *ready to execute*, then no priority-8 or lower task will get scheduled on that same CPU. In other words, the priority-9 tasks will starve all lower priority tasks.

Ready to execute

It is important to understand what 'ready' means to FreeRTOS. A task is *ready* when it is not blocked waiting for something, whether it is an event, an entry pushed into a *queue* or a *mutex* to become unlocked (we will discuss the *mutex* concept later in this series). A task that is ready to execute is inserted into the scheduler's *ready list* according to its priority and is executed when its turn comes. Because it is a priority sorted list, the highest-priority tasks are considered first.

Tasks with equal priorities are scheduled using a *Round-Robin* approach. Three 'ready' tasks at priority-9 (a, b and c) will take turns:

- task9a
- task9b
- task9c
- task9a
- task9b
- etc.

Unless they become blocked, this continues forever. Only a higher-priority task can pre-empt them. For example, the high-priority ESP32 task named *idc1* (for CPU 1), may pre-empt your priority-9 tasks to take care of some business. Once the *idc1* task becomes *not ready* again, the priority-9 tasks resume where they left off.

Here are some examples of how a FreeRTOS task becomes *not ready*:

- sleep or delay for a time (waiting for a timer);
- waiting for a mutex or semaphore;
- waiting to receive a message from an empty queue;
- waiting to insert a message into a full queue;
- waiting for a FreeRTOS event or event group;
- waiting for an I/O to complete;
- suspended (either by the task itself, or by another task).

One of the ways to become *blocked* is to wait to receive a message from an empty queue. When empty, there is nothing for that task to do so the scheduler removes that task from the ready list and searches for others to run. Only tasks on the 'ready' list will be considered. If no tasks are found, the system's idle task is run instead.
Notice that `taskYIELD()` function call is not one of the reasons listed. When a task yields control, either by exhausting its time slice or by voluntarily yielding by calling `taskYIELD()`, control returns to the FreeRTOS scheduler so that it can choose another task to run for the next slice. Yielding is not blocking because these tasks remain ready to execute and will be given CPU again at the next opportunity.

ESP-IDF FreeRTOS SMP changes

FreeRTOS was designed for single-CPU microcontrollers. Because the ESP32 consists of a dual-CPU arrangement (except ESP32-S2), Espressif customized the scheduler component. As review, the following ESP32 CPUs are present:

- CPU 0 known as the PRO_CPU (Protocol CPU);
- CPU 1 known as the APP_CPU (Application CPU);

Espressif states that the "two cores are identical in practice and share the same memory".
To support symmetric multiprocessing (SMP), they state that the "scheduler will skip tasks when implementing Round-Robin scheduling between multiple tasks in the Ready state that are of the same priority". This comes from the limitation of using a 'ready' list designed for a single CPU, on a platform that has two [2].
The problem that they faced was that when a CPU required a task context change (to run the next ready task), the CPU has only one task ready list to search. So if the current list index points to ready tasks for the other CPU, then those entries have to be skipped until an entry for the required CPU can be found. This can make the Round-Robin scheduling less than perfect. The bottom line for the developer is that the Round-Robin scheduling is not completely fair in the dual-CPU ESP32. For many projects, this will not be a noticeable but if it does become problematic, there are ways to code around it. Just be aware of this in your task planning.

Demonstration

An Arduino demonstration program is available for the Lolin ESP32 OLED Display Module (**Figure 1**). By changing a few macros in the program, you'll be able to alter task priorities of four different tasks within it. The program is designed to display three inchworms, which inch (wiggle) back and forth along the horizontal dimension of the OLED. Each of the inchworms will only hump along if the driving tasks get CPU time. CPU starved tasks will leave the worm sitting still or moving slowly.
Each worm is driven by a task that eats CPU time and then sends a message to the fourth task. This fourth task is responsible for making that worm inch and be displayed.
The code for drawing and managing the state of the inchworm, is defined in the `InchWorm` class (not shown here). For this article, we'll simply focus upon the effect of the `InchWorm::draw()` method for each worm. Each instance of the InchWorm class manages its own state and progress. The display and worm instances are declared in the program as follows:

```
static Display oled;
static InchWorm worm1(oled,1);
static InchWorm worm2(oled,2);
static InchWorm worm3(oled,3);
```

Each worm takes a C++ reference (like a C pointer) to the display class in the first argument and the number of the worm as the second. The reference to the display allows for a future enhancement like the support of multiple displays. The worm number determines where on the OLED it is displayed (1, 2 and 3 are first, middle and bottom lines respectively).
The task behind each worm, is simply a CPU time wasting loop and a message send call:

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;

  for (;;) {
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__("nop");
    xQueueSendToBack(qh,&worm,0);
    // vTaskDelay(10);
  }
}
```

It is important to leave the `vTaskDelay()` function commented out for now. It will be used in a later experiment.
The same task function is used for all three inchworm tasks, with the argument named `arg` specifying which instance of the worm that we want to wiggle. The address of the worm is converted from a void pointer and stored in the local variable `worm`. It is only used within this task to be sent as a message to indicate to the display task (main task) which worm to wiggle.
Note that when `xQueueSendToBack()` is called in this demonstration, the time-to-wait parameter has been specified as zero (argument three). This directs FreeRTOS to queue if it can but immediately fail if the queue is full. This is intentional because if the queue becomes full, we don't want our inchworm task to block its execution. The task must not release the CPU for this demonstration, so it can truly monopolize the CPU.
The outer `for` loop has the task performing its operations forever. The inner CPU time wasting `for` loop executes a *no operation* (`nop`) operation 800,000 times. The `__volatile__` keyword prevents the compiler from optimizing this loop statement away. Despite what the compiler might think, we really do want to do this wasteful thing.
Upon completion of the time wasting loop, we send the address of the worm to be wiggled to the queue identified by handle `qh`. Once the message is received by the display task, it will cause our worm to be advanced and movement displayed.
The main Arduino `loop()` task is used as the display task to perform the worm wiggling:

```
void loop() {
  InchWorm *worm = nullptr;

  if ( xQueueReceive(qh,&worm,portMAX_DELAY) )
    worm->draw();
}
```

This loop blocks execution until one of the tasks sends the address of the worm to be drawn. Once that class pointer is received, the `InchWorm::draw()` method is invoked to draw the worm and advance it.
The `setup()` function is illustrated in **Listing 1**, showing how the three worm tasks and the queue are created.

Changing priority

FreeRTOS permits a task to change its own or another task's priority using the `vTaskPrioritySet()` function. By default the task invoking `setup()` and `loop()` runs at priority level 1 (these functions are called by the same main task). For this demonstration we need that priority to be higher than the other three worm tasks. The setup() function alters the priority of its own task as follows:

Listing 1 – The setup() function.

```
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Which CPU?
  oled.init();
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
  qh = xQueueCreate(4,sizeof(InchWorm*));

  // Draw at least one worm each:
  worm1.draw();
  worm2.draw();
  worm3.draw();

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    "worm1",    // Task name
    3000,       // Stack size
    &worm1,     // Argument
    WORM1_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    "worm2",    // Task name
    3000,       // Stack size
    &worm2,     // Argument
    WORM2_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    "worm3",    // Task name
    3000,       // Stack size
    &worm3,     // Argument
    WORM3_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);
}
```

```
static int app_cpu = 0; // Updated by setup()
...
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Which CPU?
  ...
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
```

As shown, the `setup()` function obtains its own task handle by calling `xTaskGetCurrentTaskHandle()` and storing it in `h`. By changing the main task priority in the call to `vTaskPrioritySet()`, the task priority used by `loop()` is also affected. This is an example of how task priorities can be adjusted.
In the first experiment, the worm tasks are assigned task priorities 9, 8, and 7. This requires that our display (main) task to be at priority 9 or above (we will use 10). If this were not done, the main task `loop()` will starve of CPU and be unable to animate the inchworms.

Which CPU?

From the `setup()` snippet shown, another ESP32 API function named `xPortGetCoreID()` was illustrated to discover which CPU the application is running on. This is assigned to static variable `app_cpu` in the program so that the code knows which CPU to create new tasks for. For the dual core ESP32, the value of `app_cpu` will be 1 (run on CPU 1 in a dual-core configuration). For single-CPU platforms, `app_cpu` will be set to zero. Coding it this way normally allows it to portably run on single or dual platforms.

This particular demonstration however, will not function well on a single-CPU platform because of the way the CPU is monopolized. That will trigger the watchdog timer and cause resets. But the technique of using `xPortGetCoreID()` does illustrate how portability can be achieved for other applications.

Demo configuration

The demonstration source code is available at [3]. At the top of the demonstration program are macro definitions, which configure each experiment:

```
// Worm task priorities
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7

// loop() must have highest priority
#define MAIN_TASK_PRIORITY  10
```

Initially leave those as shown for the first experiment.

Custom OLED display

If you're not using the recommended Lolin ESP32 with its built-in OLED, your custom display settings can reconfigured here:

```
Display(
    int width=128,
    int height=64,
    int addr=0x3C,
    int sda=5,
    int scl=4);
```

If your settings are correctly configured, the OLED should immediately turn white upon program initialization. Otherwise, recheck the connections and settings.

Demonstration 1

Using the downloaded code, simply compile, flash and run the application. Your OLED should immediately display white, with three black inch worms drawn (see **Figure 2**).
The configuration (again) for this experiment is:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

This configuration will cause the top worm to wiggle its way across the top, while the lower two remain still. The question is: why don't the middle and bottom worms move?

```
                    _¯_
_¯_
_¯_
```

Recall that we left the main task at priority 10. So it enjoys the highest priority in our application set of tasks. The first worm, which displays on the top line of the OLED was able to progress because it was the only CPU consuming task able to run. This priority-9 task is able to execute because the priority-10 display task performs I/O to the OLED and then waits for messages to arrive in the message queue (becomes blocked). When the display task is blocked, other lower priority tasks are able to schedule.
The priority-8 and -7 tasks (for middle and bottom worms) are starved of CPU and never get executed because the priority-9 task completely monopolizes the CPU. This is the nature of real-time scheduling within FreeRTOS. Unlike Linux or Windows, lower priority tasks are not given a chance to execute.

Demonstration 2

For the second experiment, modify the configuration to give the three worms all the same priority but leave the main display task at priority 10. Set all three to the same priority of 9, 8 or 7. I'll use 9 here:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  10
```

When you recompile and reflash the ESP32, what did you observe?

```
        _¯_
        _¯_
        _¯_
```

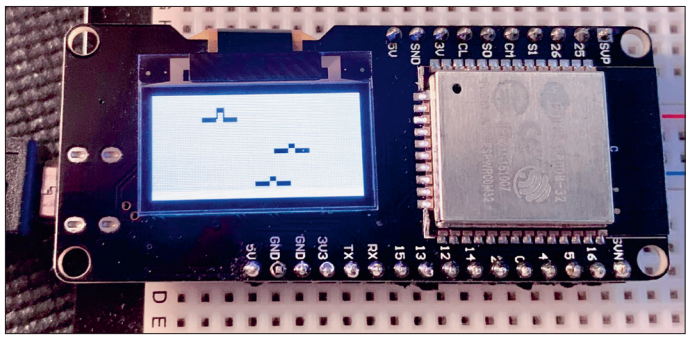

Figure 2: When tasks are executed, the demo worms move.

They will march across the screen at the same pace (or nearly so). When the demonstration is allowed to run long enough, some worms might get ahead of the others by a little bit.

Demonstration 3

In this experiment, modify the configuration to give the three worms all the same priority (as in the last demonstration) and set the main display task to that same priority. I'll use priority-9 for all of these tasks:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  9
```

After recompiling, reflashing and running the code, what did you observe? Was there a difference? Why are they progressing at different rates?

```
    _¯_
          _¯_
                        _¯_
```

When I run this, the bottom worm seems to get the most CPU (i.e. wiggles the fastest). The top worm moves the slowest. Again, the Espressif noted limitation of round-robin unfairness is to blame for this. Ideally, the display task should only steal a little CPU while drawing the inch worm. Otherwise, the remaining CPU time should be equally shared among the three other tasks driving the worms.
Yet we see that the scheduling is unbalanced. Both CPUs are responding to timer and other interrupts. The flawed scheduler code is responsible for disrupting the fairness of Round-Robin scheduling.

Demonstration 4

Each demonstration so far has had each worm task consume as much CPU time as it can muster. How does the behaviour change if we introduce a small delay (to block) within the loop? Reset the configuration so that the main display task has priority 10, and each of the worm tasks have priorities 9, 8 and 7 respectively:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
```

```
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

Then uncomment the line where `vTaskDelay()` is called so that the task loop looks like this:

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;

  for (;;) {
    ...
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__("nop");
    xQueueSendToBack(qh,&worm,0);
    vTaskDelay(10); // Uncommented
  }
}
```

Now each worm task will consume CPU, try to queue up a worm and then block for 10 milliseconds. Compile, flash and run this example. What did you observe?
The top worm will move the fastest and the bottom worm will move the slowest. The top worm with priority-9 gets first crack at the CPU due to its high priority (while the display task is blocked). When the worm task is blocked in the `vTaskDelay(10)` call, the next lower priority task (the middle worm) gets to consume some CPU and it eventually calls `vTaskDelay(10)`. This in turn allows the even lower, priority-7 task to get some cycles. This has a trickle down effect, dividing up CPU from highest to lowest levels.
But note that the priority-8 and -7 tasks do get pre-empted whenever the higher priority-9 task becomes ready again. This is why the top worm moves the fastest. The middle worm can sometimes pre-empt the priority-7 task, so it tends to be faster than the bottom worm.

More experiments

What happens if you make the `vTaskDelay()` time much longer than 10 milliseconds? Try to imagine the answer and then run it. Why did you get that result? What happens if you reduce the delay time to a 1-millisecond delay? These explorations are left for the reader.

Priority configuration

While we have not yet covered interrupt use within the ESP32, be aware of the header file named *FreeRTOSConfig.h*, which configures priorities for the platform, found here:

```
$IDF_PATH/components/freertos/include/freertos/
FreeRTOSConfig.h
```

The header defines the following priority macro values. The compiled values are shown:

```
configMAX_PRIORITIES = 25
configKERNEL_INTERRUPT_PRIORITY = 1
configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
```

The first macro defines the maximum number of priorities available. This means that valid priority numbers range from 0 to 24. The second macro defines the priority used by the kernel itself for *interrupts*. Connected with this is the third macro, which sets the highest priority used by kernel interrupts. Any FreeRTOS API call made from **within** an Interrupt Service Routine (ISR) must only call FreeRTOS API functions with names ending in `FromISR()`. Further, with the values shown, those functions can only be called from interrupt task priorities 1 to 3 inclusive. If no `FromISR()` calls are made, the ISR may freely operate at priorities 4 through 24 inclusive.

Summary

What can we conclude from these experiments? What may have seemed like a simple concept of priority was not so simple after all. The consequence is that if your task priorities are not well planned, there can be surprises — some tasks can become CPU starved. We haven't discussed watchdog timers yet but this impacts them also. For example if the watchdog timer triggers in CPU 0, then your ESP32 will reset and restart. For the dual-core ESP32, there is the additional issue that Round-Robin scheduling at the same priority level can lead to unequal execution time. This can be problematic in some applications and yet be problem free in others. The problem depends upon the nature of your 'system'.
For many applications, you can simply create tasks to run at priority 1. This is the priority configured for the Arduino `setup()` and `loop()` task. Higher-priority tasks can safely be utilized if they block on a queue, semaphore or some other event. When a task blocks or is suspended, the CPU is shared with other equal or lower priority tasks. An application with properly configured task priorities will operate like a well oiled machine. ⏮

(191195-01)

Web Links

[1] Practical ESP32 Multitasking, Elektor Magazine 1/2020: www.elektormagazine.com/190182-01
[2] Symmetric Multiprocessing: https://thc420.xyz/esp-idf/file/docs/en/api-guides/freertos-smp.rst.html
[3] Project source code: https://github.com/ve3wwg/esp32_freertos/blob/master/priority-worms1/priority-worms1.ino