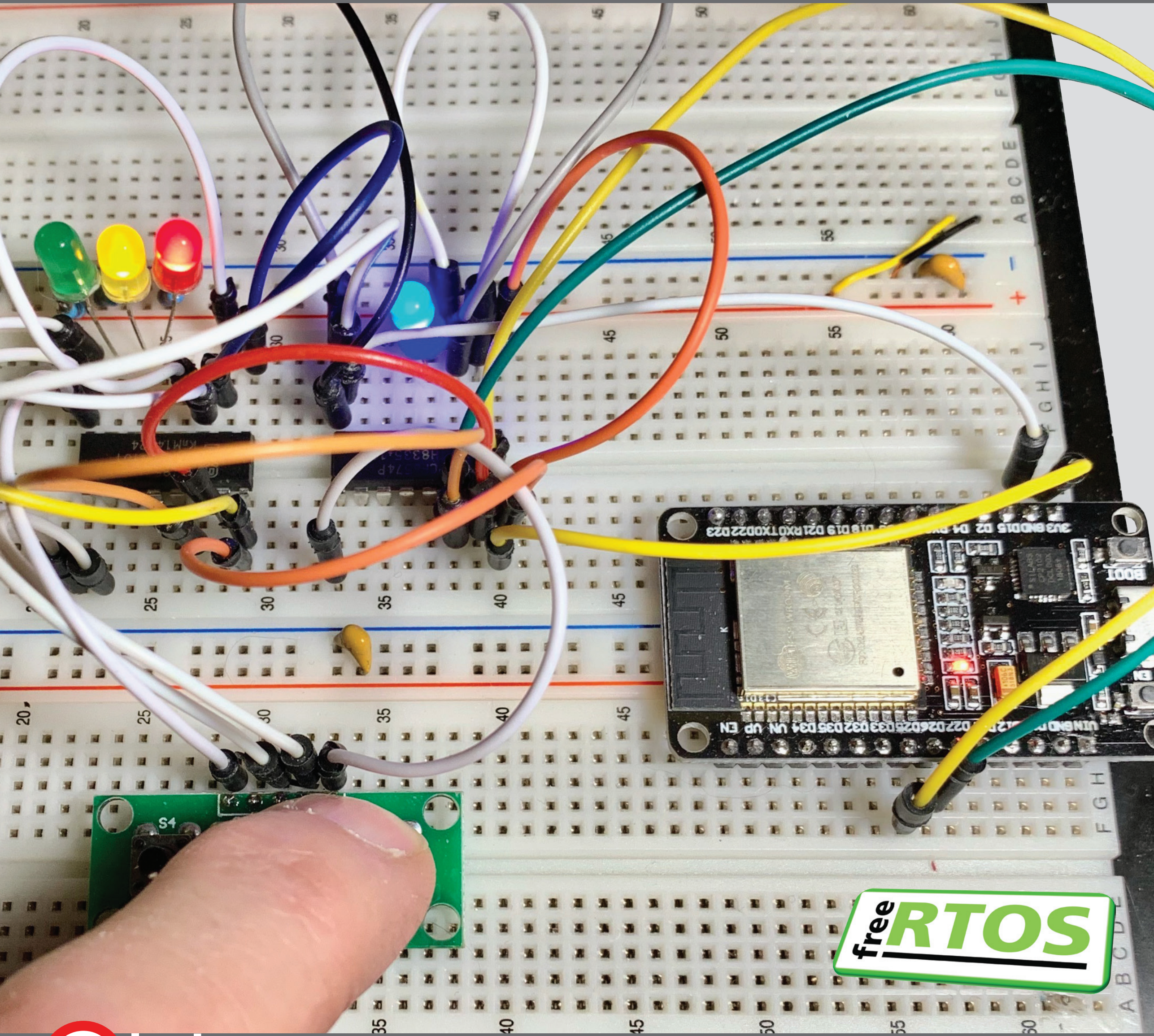
FreeRTOS for ESP32-Arduino

Practical Multitasking Fundamentals



elektor

Warren Gay

FreeRTOS for ESP32-Arduino

Practical Multitasking Fundamentals

Warren Gay

LEARN DESIGN SHARE

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
78 York Street, London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344

● British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

● **ISBN 978-1-907920-93-6**

© Copyright 2020: Elektor International Media b.v.
Prepress Production: D-Vision, Julian van den Berg
First published in the United Kingdom 2020

Chapter 1 • Introduction

In recent times, the development of System on a Chip (Soc) has lead to the popular use of microcontrollers. Many products sold today will have one or more microcontrollers found inside. Their small size, low cost, and increasing capabilities make them very compelling. Beginning in 2005, the Arduino project made microcontrollers more accessible to students by simplifying the programming environment.[1] Since then, hobbyists and engineers alike have exploited its capabilities.

More recently, FreeRTOS within the Arduino software framework has been introduced on some platforms. Why is FreeRTOS beneficial? What problems does it solve? How can FreeRTOS be leveraged by your project? These are some of the questions answered in this book with demonstrations.

Not all Arduino hardware platforms support FreeRTOS. The RTOS (Real-Time Operating System) component requires additional resources like SRAM (Static Random Access Memory) and a stack for each task. Consequently, very small microcontrollers won't support it. For larger microcontrollers that do, a rich API (Application Programming Interface) is available to make writing your application easier and more powerful.

The Need for RTOS

The general approach used on small AVR (ATmel) devices is to poll for events and respond. A program might test for button presses, incoming serial data, take temperature readings, and then at the right time, produce a result like closing relays or sending serial data. That polling approach works well enough for small projects.

As the number of input events and conditions increases, the complexity tends to multiply. Managing events by polling requires an ever-increasing management of state. Well designed programs may, in fact, implement a formal "state machine" to organize this complexity.

If instead, the same program was split into independently executing subprograms, the problem becomes much simpler to manage. Within FreeRTOS, these are known as tasks. The button press task could examine the GPIO input and debounce it. It becomes a simple loop of its own, producing an event only when the debounced result indicates that the button was pressed. Likewise, the serial input task operating independently can loop while receiving characters until an end of line character was encountered. Once the serial data was decoded, the interpreted command could signal an event. Finally, the master task, receiving both the button press and command events from other tasks can trigger an action event (like the closing of relays). In this manner, a complex application breaks down into smaller tasks, with each task focusing on a subset of the problem.

How are tasks implemented? In the early years of computing, mainframes could only run one program at a time. This was an expensive way to use a computer that occupied the size of a room. Eventually, operating systems emerged, with names like the Time Sharing Option (TSO), which made it possible to share that resource with several users (all running different programs). These early systems gave the illusion of running multiple programs at the same time by using a trick: after the current time slice was used up, the program's registers were saved, and another program's registers were reloaded, to resume the suspended program. Performed many times per second, the illusion of multiple programs running at once was complete. This is known as *concurrent execution* since only one program is running at any one instant.

A similar process happens today on microcontrollers using an RTOS. When a task starts, the scheduler uses a hardware timer. Later, when the hardware timer causes an interrupt, the scheduler suspends the current task and looks for another task to resume. The chosen task's registers are restored, and the new (previously suspended) task resumes. This concurrent execution is also known as *preemptive scheduling* because one task preempts another when the hardware timer interrupts.

Preemptive scheduling is perhaps the main reason for using FreeRTOS in today's projects. Preemptive scheduling permits concurrent execution of tasks, allowing the application designer to subdivide complex applications without having to plan the scheduling. Each component task runs independently while contributing to the overall solution.

When there are independent tasks, new issues arise. How does a task safely communicate an event to another task? How do you synchronize? How do interrupts fit into the framework? The purpose of this book is to demonstrate how FreeRTOS solves these multitasking related problems.

FreeRTOS Engineering
It would be easy to underestimate the design elegance of FreeRTOS. I believe that some hobbyists have done as much in forums. Detractors talk about the greater need for efficiency, less memory, and how they could easily implement their routines instead. While this may be true for trivial projects, I believe they have greatly underestimated the scope of larger efforts.

It is fairly trivial to design a queue with a critical section to guarantee that one of several tasks receives an item atomically. But when you factor in task priorities, for example, the job becomes more difficult. FreeRTOS guarantees that the highest priority task will receive that first item queued. Further, if there are multiple tasks at the same priority, the first task to wait on the queue will get the added item. Strict ordering is baked into the design of FreeRTOS.

The mutex is another example of a keen FreeRTOS design. When a high priority task attempts to lock a mutex that is held by a lower priority task, the later's priority is increased temporarily so that the lock can be released earlier, to prevent deadlocks. Once released, the task that was holding the mutex returns to its original priority. These are features that the casual user takes for granted.

The efficiency argument is rarely the most important consideration. Imagine your application written for one flavour of RTOS and then in another. Would the end-user be able to tell the difference? In many cases, it would require an oscilloscope measurement to note a difference.

FreeRTOS is one of several implementations that are available today. However, it’s free status and its first-class design and validation make it an excellent RTOS to study and use. FreeRTOS permits you to focus on your *application* rather than to recreate and validate a home-baked RTOS of your own.

Hardware
To demonstrate the use of the FreeRTOS API, it is useful to concentrate on one hardware platform. This eases the requirements for the demonstration programs. For this reason, the Espressif ESP32 is used throughout this book, which can be purchased at a modest cost. These devices have enough SRAM to support multiple tasks and have the facilities necessary to support preemptive scheduling. Even more exciting, is the fact that these devices can also support WiFi and TCP/IP networking for advanced projects.

Dev Boards
While almost any ESP32 module could be used, the reader is encouraged to use the "dev board" variety for this book. The non-dev board module requires a TTL to serial device to program its flash memory and communicate with. Be aware that many TTL to serial devices are 5 volts only. To prevent permanent damage, these should *not* be used with the 3.3 volt ESP32. TTL to serial devices can be purchased, which do support 3.3 volts, usually with a jumper setting.

The dev boards are much easier to use because they include a USB to serial chip onboard. They often use the chip types CP2102, CP2104, or CH340. Dev boards will have a USB connector, which only requires a USB cable to plug into your desktop. They also provide the necessary 5 volts to 3.3-volt regulator to power your ESP32. GPIO 0 is sometimes automatically grounded by the dev board, which is required to start the programming. The built-in USB to serial interface makes programming the device a snap and permits easy display of debugging information in the Arduino Serial Monitor. Dev boards also provide easy GPIO access with appropriate labels and are breadboard friendly (when the header strips are added). The little extra spent on the dev board is well worth the convenience and the time it will save you.

One recommended unit is the ESP32 Lolin with OLED because it includes the OLED display. It is priced a little higher because of the display but it can be very useful for end user applications. Most ESP32 devices are dual-core (two CPUs), and the demonstrations in this book assume as much.

If you are determined to use the nondev board variety, perhaps because you want to use the ESP32CAM type of board, then the choice of USB to TTL serial converter might be important. While the FT232RL eBay units offer a 3.3-volt option, I found that they are problematic for MacOS (likely not for Windows). If the unit is unplugged or jiggled while the device is in use, you lose access to the device, and replugging the USB cable doesn’t help. Thus it requires the pain of rebooting and is, therefore, best avoided.

Table 1-1 summarizes the major Espressif product offerings that will populate various development boards. When buying, zoom in on the CPU chip in the photo for identifying marks. There are other differences between them in terms of peripheral support etc., not shown in the table. Those details can be discovered in the Espressif hardware PDF datasheets. All examples in this book assume the dualcore CPU to run without modification. The demonstrations can be modified to work on a single core unit but when learning something new, it is best to use tested examples first.

Series	Cores	CPU Clock	SRAM + RTC	Marking
ESP32	2	80 to 240 MHz	520kB+16kB	ESP32-D0WD
				ESP32-D0WDQ6
				ESP32-D2WD
	1		520kB+16kB	ESP32-S0WD
ESP32-S2	1	240 MHz	320kB+16kB	

Table 1-1. Major Espressif Product Offerings

ESP8266

The hardware of the ESP8266 is quite capable of supporting FreeRTOS. If you use the Espressif ESP-IDF (Integrated Development Framework), you can indeed make use of the FreeRTOS API there. Unfortunately, the Arduino environment for the ESP8266 does not make this available, even though it has been used internally in the build.

To keep things simple for students familiar with Arduino therefore, this book is focused on the dualcore ESP32. What is described for FreeRTOS in this book can also be applied to the ESP-IDF for both the ESP8266 and the ESP32 devices.

FreeRTOS Conventions

Throughout this book and in the Arduino source code, I'll be referring to FreeRTOS function names and macros using their naming conventions. These are outlined in the FreeRTOS manual, which is freely available for download as a PDF file.[2] These are described in Appendix 1 of their manual.

While I am not personally keen on this convention, it is understood that the FreeRTOS authors were thinking about portability to many different platforms. Knowing their conventions helps when examining the reference API. The following two data types are used frequently:

- TickType_t – For the Espressif platform, this is a 32-bit unsigned integer (uint32_t), which holds the number of system ticks.

- BaseType_t – For the Espressif platform, this is defined as a 32-bit unsigned integer (uint32_t). The type is chosen to be efficient on the hardware platform being used.

Variable Names
The variable names used in FreeRTOS examples and arguments, use the following prefixes:

- 'c' - char type
- 's' - short type
- 'l' - long type
- 'x' - BaseType_t and any other types not covered above

A variable name is further prefixed with a 'u' to indicate an unsigned type. If the value is a pointer, the name is prefixed with 'p'. An unsigned character variable would use the prefix 'uc', while a pointer to a char type, would be 'pc'.

Function Names
Function names are prefixed with two components:

- The data type of the value returned
- The file that the function is defined in

These are some of the examples they provide:

- vTaskPrioritySet() returns a void and is defined within FreeRTOS file task.c.
- xQueueReceive() returns a variable of type BaseType_t and is defined within FreeRTOS file queue.c.
- vSemaphoreCreateBinary() returns a void and is defined within FreeRTOS file semphr.h.

In this context, the prefix 'v' means that the function returns void.

Macro Names
Macro names are given in uppercase, except for a prefix that indicates where they are defined (Table 1-2).

Prefix	File	Example
port	portable.h	portMAX_DELAY
task	task.h	taskENTER_CRITICAL()
pd	projdefs.h	pdTRUE
config	FreeRTOSConfig.h	configUSE_PREEMPTION
err	projdefs.h	errQUEUE_FULL

Table 1-2. Macro name conventions used by FreeRTOS.

Header Files
Normally when using FreeRTOS there are #include statements required. Within the ESP32 Arduino programming environment, these are already provided internally. However, when you use the ESP-IDF or a different platform, you will need to know about the header files

listed in Table 1-3. Because they are not required in the Arduino code, they are only mentioned here.

Header File	Category	Description
FreeRTOS.h	First	Should be the first file included.
FreeRTOSConfig.h	Included by FreeRTOS.h	Not required when FreeRTOS.h has been included.
task.h	Tasks	Task support
queue.h	Queues	Queue support
semphr.h	Semaphores	Semaphore and mutex support
timers.h	Timers	Timer support

Table 1-3. FreeRTOS Include Files

Arduino Setup

This book uses the installed Arduino IDE rather than the newer web offering. If you've not already installed the Arduino IDE and used it, you might want to do so now. If you're using MacOS and recently upgraded to Catalina, you will also need to update your Arduino software. The IDE is downloadable from:

https://www.arduino.cc/en/main/software

Click the appropriate platform link for the install. At the time of writing, the website lists the following choices:

- Windows Installer, for Windows XP and up
- Windows Zip file for non-admin install
- Windows App. Requires Win 8.1 or 10.
- Mac OS X 10.8 Mountain Lion or newer.
- Linux 32 bits.
- Linux 64 bits.
- Linux ARM 32-bits.
- Linux ARN 64-bits.

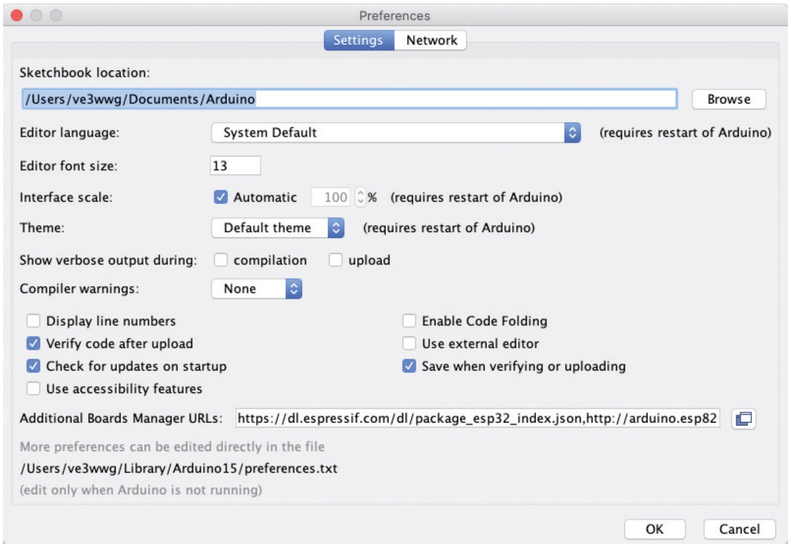
Install guidance or troubleshooting is best obtained from that Arduino website. Normally, the IDE installs without problems.

ESP32 Arduino

To add ESP32 support to the Arduino IDE, open File->Preference (see Figure 1-1), Arduino->Preferences for MacOS, and add:

https://dl.espressif.com/dl/package_esp32_index.json

to your "Additional Boards Manager URLs" text box. If you already have something in there, then separate the addition with a comma (,).


Preferences

Settings | Network

Sketchbook location:
/Users/ve3wwg/Documents/Arduino — Browse

Editor language: System Default (requires restart of Arduino)

Editor font size: 13

Interface scale: [✓] Automatic 100 % (requires restart of Arduino)

Theme: Default theme (requires restart of Arduino)

Show verbose output during: [ ] compilation [ ] upload

Compiler warnings: None

- [ ] Display line numbers
- [✓] Verify code after upload
- [✓] Check for updates on startup
- [ ] Use accessibility features
- [ ] Enable Code Folding
- [ ] Use external editor
- [✓] Save when verifying or uploading

Additional Boards Manager URLs: https://dl.espressif.com/dl/package_esp32_index.json,http://arduino.esp82

More preferences can be edited directly in the file
/Users/ve3wwg/Library/Arduino15/preferences.txt
(edit only when Arduino is not running)

OK | Cancel

Figure 1-1. File->Preference dialog.

Next choose Tools->Board->Board Manager (Figure 1-2):

- Then search for "ESP32" and click install (or update), if supporting the ESP32.

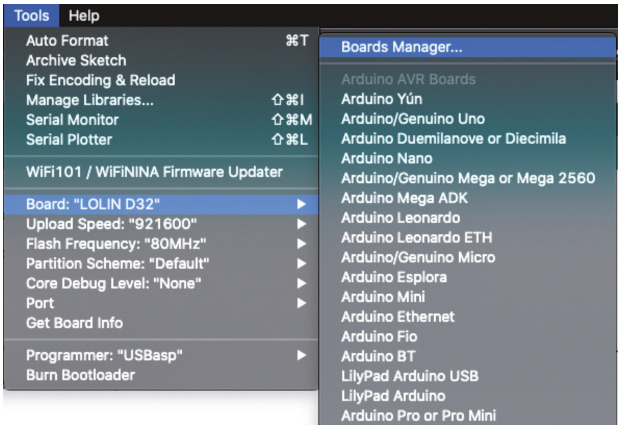
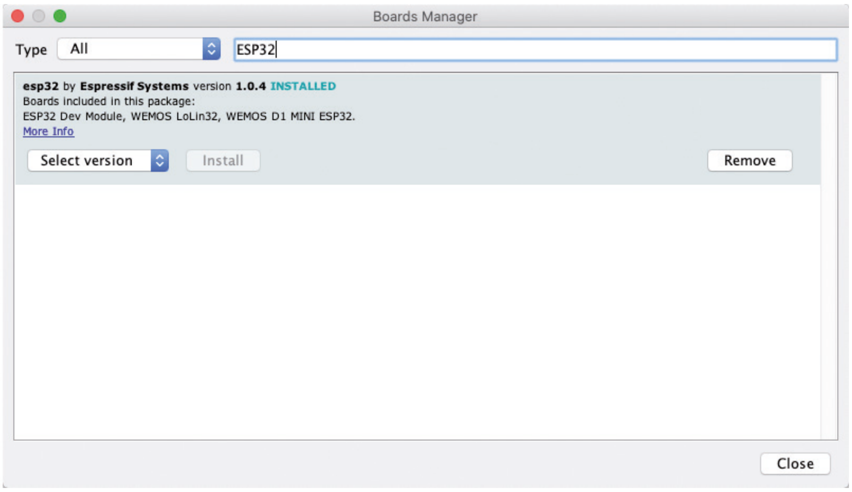


Figure 1-2. File->Board->Boards Manager menu selection.

Figure 1-3 shows the dialog after the ESP addition has been installed. That should be all you need to do, to add Espressif support to your Arduino IDE.



esp32 by Espressif Systems version 1.0.4 INSTALLED
Boards included in this package:
ESP32 Dev Module, WEMOS LoLin32, WEMOS D1 MINI ESP32.
More Info

Figure 1-3. Boards Manager after the ESP addition is installed.

ESP Related Arduino Resources
If IDE issues arise, then search the resources found at

https://www.arduino.cc/

If the problem is Espressif related, then there is the following Arduino GitHub page:

https://github.com/espressif/arduino-esp32

C and C++
It surprises some to learn that the Arduino framework uses the C++ language. This may be deemphasized to prevent scaring prospective students – a common perception is that C++ is difficult. But C++ is increasingly finding its way into embedded programming circles because of its advantages of stronger type checking among other advantages. The career student is therefore encouraged to embrace it.

This book will use a dabbling of C++ language features when it is useful, instructive, or just plain preferred. One trivial example is the keyword nullptr is favoured over the old C macro NULL. Where C++ classes are used, they are simple objects. No C++ template programming is used in this book, so there is no need for fear and loathing.

There is one area that Arduino users will bump into when looking at Espressif provided example code. Most of their examples are written in C. The C language structure initialization *differs* from C++, although there are efforts working towards harmonization. Listing 1-1 shows a fragment of an Espressif C language wifi scan example. Notice the initialization syntax of the structure named wifi_config.

```
static void wifi_scan(void)
{
...
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = DEFAULT_SSID,
            .password = DEFAULT_PWD,
            .scan_method = DEFAULT_SCAN_METHOD,
            .sort_method = DEFAULT_SORT_METHOD,
            .threshold.rssi = DEFAULT_RSSI,
            .threshold.authmode = DEFAULT_AUTHMODE,
        },
    };
...
}
```

Listing 1-1. Espressif examples/wifi/scan/main/scan.c fragment.

Members like .ssid are set to initialization values using the C language syntax. This style of initialization is not yet supported by C++. Given that Arduino code is C++, you cannot copy and paste C language structure initialization code into your program and expect it to compile.

Listing 1-2 shows *one* way it can be reworked in C++ terms (advanced users can also use the extern "C" approach). First, clear the structure completely to zero bytes by using the memset() function. Once cleared, the individual members can be initialized as required.

```
static void wifi_scan(void)
{
...
    wifi_config_t wifi_config;

    memset(&wifi_config,0,sizeof wifi_config);
    wifi_config.sta.ssid = DEFAULT_SID;
    wifi_config.sta.password = DEFAULT_PWD;
...
    wifi_config.sta..threshold.authmode = DEFAULT_AUTHMODE;
...
}
```

Listing 1-2. Function wifi_scan() Converted to C++ initialization.

FreeRTOS and C++

FreeRTOS is written in the C language to give it the greatest portability among microcontroller platforms and compiler tools. Yet it is quite useable from C++ code since the compiler is informed from the header files that the FreeRTOS API functions are C language declarations. Because these are C language calls, some C++ restrictions naturally follow. For example, when using the FreeRTOS queue, you cannot add an item that is a C++ object, requiring constructors or destructors. The data item must be POD (Plain Old Data). This is understandable when you consider that the C language doesn't support class objects, constructors, or destructors.

Arduino FreeRTOS Config

The Arduino environment that is built for your ESP32 uses a predefined FreeRTOS configuration to declare the features that are supported and configure certain parameters. These macros are already included for the Arduino but other environments like ESP-IDF, require including the header file FreeRTOS.h. Many of the configured values for the ESP32 Arduino are provided in Table 1-4 for your convenience. The detailed meaning of these values is documented by the FreeRTOS reference manual, which is freely available online.

Macro	Value	Notes
configAPPLICATION_ALLOCATED_HEAP	1	ESP32 defined heap.
configCHECK_FOR_STACK_OVERFLOW	2	Check for stack overflow by initializing stack with a value at task creation time.
configESP32_PER_TASK_DATA	1	Per task storage facility
configEXPECTED_IDLE_TIME_BEFORE_SLEEP	2	
configGENERATE_RUN_TIME_STATS	0	Disabled
configIDLE_SHOULD_YIELD	0	Disabled
configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS	0	Disabled
configMAX_TASK_NAME_LEN	16	Maximum name string length
configMINIMAL_STACK_SIZE	768	Idle task stack size (bytes)
configQUEUE_REGISTRY_SIZE	0	Queue registry not supported
configSUPPORT_DYNAMIC_ALLOCATION	1	Dynamic memory supported
configSUPPORT_STATIC_ALLOCATION	No support	
configTASKLIST_INCLUDE_COREID	0	Disabled
configUSE_ALTERNATIVE_API	0	Disabled
configUSE_APPLICATION_TASK_TAG	0	Disabled
configUSE_COUNTING_SEMAPHORES	1	Counting semaphores enabled
configUSE_MALLOC_FAILED_HOOK	0	Disabled
configUSE_MUTEXES	1	Mutexes enabled
configUSE_NEWLIB_REENTRANT	1	Reentrancy for newlib enabled
configUSE_PORT_OPTIMISED_TASK_SELECTION	0	Disabled
configUSE_QUEUE_SETS	1	Queue sets enabled
configUSE_RECURSIVE_MUTEXES	1	Recursive mutexes enabled
configUSE_STATS_FORMATTING_FUNCTIONS	0	Disabled

Macro	Value	Notes
configUSE_TASK_NOTIFICATIONS	1	Task notifications enabled
configUSE_TICKLESS_IDLE	No support	
configUSE_TIMERS	1	Timer support enabled
configTIMER_TASK_STACK_DEPTH	2048	Svc Tmr task stack size (bytes)
configTIMER_QUEUE_LENGTH	10	Depth of the command queue
configUSE_TIME_SLICING	1	Time slicing enabled (see reference manual)
configUSE_TRACE_FACILITY	0	Disabled FreeRTOS trace facilities

Table 1-4. Some ESP32 Arduino FreeRTOS configuration values.

A number of these are of special interest to Arduino users because we can determine that:

- The maximum string name for tasks and other FreeRTOS objects is 16 characters (configMAX_TASK_NAME_LEN).
- There is support for counting semaphores (configUSE_COUNTING_SEMAPHORES).
- There is support for mutexes (configUSE_MUTEXES).
- Mutex support includes recursive mutexes (configUSE_RECURSIVE_MUTEXES).
- There is support for queue sets (configUSE_QUEUE_SETS).
- There is support for task notification (configUSE_TASK_NOTIFICATIONS).
- There is support for FreeRTOS timers (configUSE_TIMERS).
- The Idle task uses a stack size of 768 bytes (configMINIMAL_STACK_SIZE).

For portability, the user can use these macros to determine the level of support available.

ESP32 Notes

This section provides a few brief reminders about ESP32 devices. The Espressif web resources and forums are the best places to get more detailed information.

Arduino GPIO References

Arduino maps "digital pin x" to a port and pin combination on some platforms. For example, digital pin 3 maps to pin PD3 on the ATmega328P. For the ESP32 Arduino environment, digital pin x maps directly to GPIO x.

Input Only

The ESP32 platform also has hardware limitations for GPIO. For example, GPIO 34 to 39 inclusive can only be used for *input.* These GPIO pins also lack the programmed pull-up resistor feature. For this reason, these inputs should always be used with external pull-up resistors for push button and switch inputs to avoid floating signals. A resistance of 10k to 50k ohm is sufficient.

Reserved GPIOs

Several ESP32 GPIO pins are reserved or are already in use by peripherals. For example, GPIO pins 6 through 11 are connected to the integrated SPI flash.

GPIO Voltage and Drive
The ESP32 device uses 3.3-volt GPIO ports and *none* are 5 volts tolerant. Inputs should never be subjected to above 3.3 + 0.6 volts (one silicon diode voltage drop). Voltages above 3.9 volts will subject the built-in protection diode to high currents and potentially destroy it. With the protective ESD (Electrostatic Discharge) diode damaged, the GPIO will be vulnerable to damage from static electricity (from the likes of the family cat). Alternatively, the ESD diode can short, causing a general malfunction of the port.

Output GPIOs have programmable current strengths, which default to strength 2. This is good for up to 20 mA.[3]

Programs
Arduino promotes the term "sketch" for their programs. I'll continue to refer to them as programs because this is the more widely accepted term. If the student pursues a career in embedded programming, he/she will most likely be using a non-Arduino framework for building *programs*. So I think it best to get comfortable with the term.

Many of the demonstrations written for this book are illustrated using the Wemos Lolin ESP32 dev board, which includes the built-in OLED. The OLED is only used by a few of the demonstration programs. Even then, some of those demonstrations can use the Serial Monitor instead. Otherwise, almost any "dev board" can be used if a reasonable complement of GPIOs is made available for you to use.

The serial interface brings with it a nagging problem for the Arduino ESP32. It would be desirable to have programs that run both with and without the USB serial interface plugged in. Yet it seems that the programs that make use of the serial interface will hang when not connected. Yet the Serial Monitor is too useful to forego for debugging and informational displays. Consequently, most demonstrations use the serial interface as provided by the ESP32 inclusion of the newlib library.[4] The first printf() call encountered, will assume a serial interface at 115,200 baud, 8 bits, no parity, and 1 stop bit. By default, the Arduino IDE will provide this in its Serial Monitor.

It may be desirable in some cases to use a demonstration program without the Serial Monitor. In that case, comment out the printf() statements and re-flash the recompiled program. If it still hangs, look for remaining uncommented printf() calls.

Graphics/Drivers Used
The main graphic driver used is for the Wemos Lolin ESP32 that has the built-in OLED. This library is found by using the Arduino Tools -> Manage Libraries and searching for "ESP8266 and ESP32 Oled Driver for SSD1306 display" by ThingPulse, Fabrice Weinberg.

TTGO ESP32 T-Display
The graphics driver used for the TTGO ESP32 T-Display unit is the driver found by the Arduino Tools -> Manage Libraries and searching for "TFT_eSPI" by Bodmer (version 2.1.4 was tested). This driver requires further editing before it can be used (see Chapter 9, Interrupts).

M5Stack
M5Stack examples in this book require that you've installed the drivers found by the Arduino Tools -> Manage Libraries and searching for "M5Stack". Choose and install the library "M5Stack" by M5Stack. Version 0.2.9 was tested in this book.

Assumptions about the Reader
This book is targeted to new and advanced Arduino users alike. The new student will discover the benefits of FreeRTOS design by wading gently into RTOS concepts and the API. The advanced user looking to become familiar with FreeRTOS can quickly familiarize themselves with the API. The emphasis was placed on the practical but some background material is provided for the benefit of new students.

Because this book is focused upon FreeRTOS, the reader is assumed to have some familiarity with the ESP32 and the Arduino API. Activities like configuring and flashing the correct board from the Arduino IDE is assumed. Those encountering difficulties in these areas are encouraged to seek help from online documentation and forums for Arduino and Espressif.

Summary
At this point, I expect that you are champing at the bit to get started. By now, your Arduino IDE has been made ready and you have some ESP32 hardware ready to play with. Let's begin that journey of discovery into FreeRTOS!

Web Resources
[1] https://en.wikipedia.org/wiki/Arduino
[2] https://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V10.0.0.pdf
[3] https://www.esp32.com/viewtopic.php?t=5840
[4] https://sourceware.org/newlib/

Chapter 2 • Tasks

Let's Multi-task!

Having portions of code executing independently as tasks within an application simplifies the design of a large complicated problem. Task support also permits selected functions to run in parallel when there is more than one CPU. This chapter will survey the FreeRTOS task support of the Arduino framework for ESP32 series devices. With a few exceptions, this material applies to other hardware platforms using FreeRTOS that you may encounter.

Preemptive Scheduling

In a single-core MCU (Microcontroller Unit), only one task can execute at any instance in time. The task that is executing runs until a hardware timer indicates that the time slice has expired. At timeout, the FreeRTOS scheduler saves the state of the current task by saving its registers. The current task is said to have been *preempted* by the timer.

The scheduler then chooses another task that is ready to run. The state of the highest priority task, which is ready to run, is restored and resumed where it left off. The duration of the time slice is small enough, that the MCU can run several tasks per second. This is known as *concurrent* processing.

The hardware timer is critical to concurrent processing because it prevents one task from monopolizing the CPU. A task stuck in a never-ending loop will not prevent other tasks from executing. Finally, preemptive scheduling allows the task function to be written as if it were the only program in the system. Programmer serendipity!

The dual-core ESP32 can execute two tasks simultaneously because of the added CPU. This potentially accomplishes twice as much work. The Espressif naming convention for its chips can be confusing, however. An ESP32 may be a single or dual-core chip. There is also the ESP32-S, which may still be a dual-core. Finally, the new ESP32-S2 is a single-core CPU. Table 2-1 lists some commonly available ESP chips by identifier.[1]

Identifier	Cores	Description
ESP32-D0WDQ6	2	Initial production release chip of the ESP32 series.
ESP32-D0WD	2	Smaller physical package variation similar to ESP32-D0WDQ6.
ESP32D2WD	2	2 MB (16 Mb) embedded flash memory variation.
ESP32S0WD	1	Single-core processor variation.
ESP32-S2	1	The newest S2 variant.

Table 2-1. ESP32 SoC Variants by CPU Identification.

The example programs in this book will assume a dual-core CPU unless otherwise designated. Many times a dual-core program can be adapted to run on a single-core processor but complicating factors like the watch-dog timer often require additional measures.

Arduino Startup
Before we review task creation and control, let's examine the Arduino environment that you inherit when your program begins. Does your program start within a task? Indeed it does! In addition to your task, other FreeRTOS tasks are executing in the background. Some of these tasks provide services such as timers, WiFi, TCP/IP, Bluetooth, etc.

Table 2-2 illustrates the FreeRTOS tasks running when the functions setup() and loop() are invoked. The task named loopTask is the main Arduino task, which calls functions setup() and loop().

Task Name	Task #	Priority	Stack	CPU
loopTask	12	1	5188	1
Tmr Svc	8	1	1468	0
IDLE1	7	0	592	1
IDLE0	6	0	396	0
ipc1	3	24	480	1
ipc0	2	24	604	0
esp_timer	1	22	4180	0

Table 2-2 – Tasks running at Arduino Startup on ESP32

The column labeled Stack represents *unused* stack bytes (every task needs its own stack). This table is sorted in reverse chronological order, by task number. The loopTask is the last task created. Missing task numbers imply that other tasks were created and have ended when their job was completed. The priority column illustrates the task priorities assigned, with zero representing the lowest execution priority.

Finally, the tasks on the dual-core ESP32 are divided between CPU 0 and CPU 1. Espressif places support tasks in CPU 0, while application tasks run on CPU 1. This keeps services for WiFi, TCP/IP and Bluetooth, etc. running smoothly without special consideration from your application. Despite this convention, it is still possible for you to create tasks in either CPU. Naturally, when using single CPU platforms, everything runs in CPU 0.

Main Task

Listing 2-1 illustrates the slightly simplified Arduino startup code. The line numbers shown in the listing are not included in the source code, but are provided for ease of reference.

```
0001: void loopTask(void *pvParameters) {
0002:
0003:   setup();
0004:   for (;;) {
0005:     loop();
0006:   }
0007: }
0008:
0009: extern "C" void app_main() {
0010:
0011:   initArduino();
0012:   xTaskCreatePinnedToCore(
0013:     loopTask,         // function to run
0014:     "loopTask",       // Name of the task
0015:     8192,             // Stack size (bytes!)
0016:     NULL,             // No parameters
0017:     1,                // Priority
0018:     &loopTaskHandle,  // Task Handle
0019:     1);               // ARDUINO_RUNNING_CORE
0020: }
```

Listing 2-1: Simplified ESP32 Arduino Startup

From this example, we can note some interesting points:

1. Note that this is a C++ startup (due to extern "C" declaration of app_main() in line 9).
2. Arduino initialization is performed by initArduino() (line 11).
3. The loopTask is created and run by the call to xTaskCreatePinnedToCore() (line 12).

The loopTask is created by invoking xTaskCreatePinnedToCore(), with several arguments. The first argument is the address of the function to be executed for the task (loopTask line 1). Once the task is created, the function loopTask() calls setup() first (line 3) and then loop() (line 5) from a forever "for" loop.

On non-ESP platforms, the task create function is likely to be named xTaskCreate() rather than xTaskCreatePinnedToCore(). The later is an Espressif extension, permitting the caller to choose the CPU for the task.

Task Demonstration
An example demonstration program is provided in Listing 2-2. This program provides a demonstration of how FreeRTOS tasks are created. It creates a task for each of three LEDs that are flashed on and off according to different timing. Figure 2-1 illustrates the circuit for almost any ESP32 device, with a single or dual CPU. Again, a "Dev board" is recommended, since the USB port simplifies the programming and use of the device.

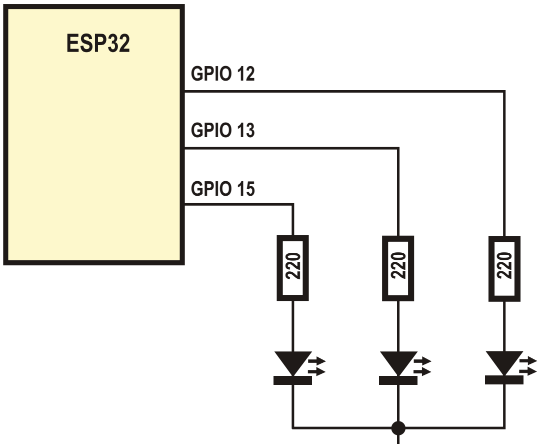


Figure 2-1. The ESP32 circuit for the program in Listing 2-2.

Lines 7, 8, or 9 can be changed if you want to move LEDs to different gpio pins. Just be aware of the ESP device limitations. For example, on the ESP32, GPIOs 32 to 36 and 39 are input only.

```
0001: // basic_tasks.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // Change the following if you want to use
0005: // different GPIO pins for the three LEDs
0006:
0007: #define LED1  12  // GPIO 12
0008: #define LED2  13  // etc.
0009: #define LED3  15
0010:
0011: struct s_led {
0012:   byte         gpio;   // LED GPIO number
0013:   byte         state;  // LED state
0014:   unsigned     napms;  // Delay to use (ms)
0015:   TaskHandle_t taskh;  // Task handle
0016: };
0017:
0018: static s_led leds[3] = {
0019:   { LED1, 0, 500, 0 },
  { LED2, 0, 200, 0 },
  { LED3, 0, 750, 0 }
};

static void led_task_func(void *argp) {
  s_led *ledp = (s_led*)argp;
  unsigned stack_hwm = 0, temp;

  delay(1000);

  for (;;) {
    digitalWrite(ledp->gpio,ledp->state ^= 1);
    temp = uxTaskGetStackHighWaterMark(nullptr);
    if ( !stack_hwm || temp < stack_hwm ) {
      stack_hwm = temp;
      printf("Task for gpio %d has stack hwm %u\n",
        ledp->gpio,stack_hwm);
    }
    delay(ledp->napms);
  }
}

void setup() {
  int app_cpu = 0;  // CPU number

  delay(500);       // Pause for serial setup

  app_cpu = xPortGetCoreID();
  printf("app_cpu is %d (%s core)\n",
    app_cpu,
    app_cpu > 0 ? "Dual" : "Single");

  printf("LEDs on gpios: ");
  for ( auto& led : leds ) {
    pinMode(led.gpio,OUTPUT);
    digitalWrite(led.gpio,LOW);
    xTaskCreatePinnedToCore(
      led_task_func,
      "led_task",
      2048,
      &led,
      1,
      &led.taskh,
      app_cpu
    );
    printf("%d ",led.gpio);
0066:   }
0067:   putchar('\n');
0068: }
0069:
0070: void loop() {
0071:   delay(1000);
0072: }
```

Listing 2-2. Program basic_tasks.ino

Program Design

Let's delve into the design of the program in Listing 2-2. A structure named s_led groups the following information elements together as a unit:

1. The GPIO number of the LED to be driven as an output (line 12).
2. The state of the LED (0=off, 1=on, line 13).
3. The delay() time to be used for this LED (line 14).
4. The handle of the created task, driving this LED (line 15). This value is populated in line 62 of the listing.

The values are initialized in the array named leds, with the array size of 3 (lines 18 to 22). The task handles are populated when the tasks are created (lines 56 to 63).

The function led_task_func() forms the task code to drive the LEDs (lines 24 to 40). Every FreeRTOS task function is required to accept one void pointer as an argument, whether it is used or not (line 24). In this program, the task expects to be started with a pointer to one of the s_led structures. Line 25 takes this void pointer and casts it to an s_led pointer, so that we can reference the structure members.

Note: it is not necessary to use the keyword "struct" when referencing the structure type in C++.

The call to delay() in line 28 is used to allow the setup() time to complete before the task gets underway. Otherwise, messages produced by the task and the setup code would jumble together in the serial monitor output. There are better ways to synchronize this but that is covered later in the book.

Lines 30 to 39 loop forever. Line 31 toggles the output state of the chosen LED (ledp->gpio), while keeping track of its state (in ledp->state). At the same time, the state is flipped, using the exclusive-or assignment operation (ledp->state ^= 1).

Note: In C++ you can use the keyword nullptr instead of the traditional C macro NULL.

Line 32 is used in this demonstration to call uxTaskGetStackHighWaterMark() with a null pointer. The calling argument normally provides the handle of the task to be queried. But nullptr implies the *current* task. The value returned is the number of stack *bytes* that are *unused* in the current stack. Low values indicate that the stack was nearly exhausted at some point. High values suggest that you could reduce the size of the stack. Stack size is specified in line 59, when the task is created. If on a subsequent loop, the stack value becomes even lower than previously reported, lines 34 to 36 will report the new value.

Finally, the task uses a unique delay time provided by structure member ledp->napms (milliseconds, from line 14). This will cause the three LEDs to flash independently at different rates. With some effort, you could program the same effect without using tasks but that would require more effort to get correct. Bask in the freedom of tasks instead!

The setup routine starts in line 42. The FreeRTOS function xPortGetCoreID() (line 47) is another Espressif addition to FreeRTOS. It is provided so that the current CPU number can be determined. The setup() code will get the value 1 when running on a dual-core platform, and zero otherwise. Espressif places most support tasks in CPU 0, leaving CPU 1 for application use. When there is only one core, all tasks run in CPU zero. This is one way you can make your Arduino code platform portable.

The loop in lines 53 to 65, configure the GPIO pins that are used to drive the LED (line 54) as well as to create their tasks (lines 56 to 63).

The for loop in line 53 needs explanation for those who are new to C++. The compiler knows about the array declaration named leds, repeated here for convenience:

```
0018: static s_led leds[3] = {
0019:   { LED1, 0, 500, 0 },
0020:   { LED2, 0, 200, 0 },
0021:   { LED3, 0, 750, 0 }
0022: };
```

The compiler knows that the array has been declared with three array elements. This permits the compiler to automatically produce a loop like:

```
for ( int x=0; x<3; ++x ) {
  s_led& led = leds[x]; // Reference to leds[x]
  ...
```

In C++, you can make the compiler do the all the dirty work and code it glibly as:

```
for ( auto& led : leds ) {
  ...
```

This is safe because the compiler will not get the array extents wrong. If you later changed the array to have a different number of members, the compiler will adjust for it. The name "led" in this new fangled for loop is a reference to the structure leds[x] (as seen in the traditional code where the reference assignment is shown explicitly). A C++ *reference* permits you to access the structure members such as led.gpio, for example.

Lines 56 to 63 call upon FreeRTOS function xTaskCreatePinnedToCore to allocate and start the task. The code that will be run, is specified by a function name led_task_func (line 57). The task starts execution the moment it is created when priority permits. An optional task (string) name is provided in line 58. This string is for our benefit and thus does not strictly need to be unique (it does need to be unique if you lookup a task by name, however).

The size of the task's stack is provided in line 59. This is in *bytes* on Espressif platforms. Line 61 specifies the task priority of 1. The lowest priority is zero but the setup()/loop() loopTask runs at priority 1. Until you familiarize with FreeRTOS priorities, I recommend that you use priority 1. The next argument expects a pointer to the storage to return the task handle in (line 62). If you don't care about the handle, nullptr/NULL can be provided instead.

Note: On most platforms, the stack size provided to FreeRTOS is the number of platform words (not bytes). For 32-bit platforms, this is 4-byte words. Espressif however, has modified FreeRTOS to expect bytes instead.

The last argument given in line 63 specifies which CPU core to run the task on. This is an Espressif enhancement, necessary for multi-core support. Table 2-3 lists your choices for this argument:

Value/Macro Name	Description
0	Protocol, or only CPU (always valid)
1	Application CPU (on a dual-core)
tskNO_AFFINITY	Apply task to any available CPU

Table 2-3. Valid CPU argument values

In the demonstration code, the current CPU core was determined in line 47. This is a safe value to provide to the xTaskCreatePinnedToCore() function. Running application code on the Protocol CPU is permissible but not always recommended. The application might interfere with the processing of TCP/IP (for WiFi) for example if it consumes too much CPU time.

Before starting the program be sure to open the serial monitor. The example monitor output is discussed in the next section.

Stack Size

Everyone needs to know how to determine the task's stack size. The reality is that unless you are willing to trace through every function that is called, and the functions that they in turn call, it is difficult to determine the worst-case needs. Stack size is critical for code that lives depend upon.

For hobby projects, it is sufficient to take a guess and try it. Start big and reduce. Once your code has run, you can find out how much reserved stack space you have remaining in each task (lines 32 to 37). If the code aborts when you run it, then you may need larger stack sizes. This procedure is not foolproof since measured usage depends upon *code coverage*.

It is possible that when the stack usage was measured, you didn't invoke functions that required larger amounts of stack.

When you run the demo program, the serial monitor should show messages like the following:

```
app_cpu is 1 (Dual core)
LEDs on gpios: 12 13 15
Task for gpio 13 has stack hwm 1616
Task for gpio 15 has stack hwm 1624
Task for gpio 12 has stack hwm 1620
Task for gpio 13 has stack hwm 556
Task for gpio 12 has stack hwm 560
Task for gpio 15 has stack hwm 564
```

Note in this example that the first time that the GPIO 13 task reported its stack high water mark (hwm), it was reported as 1616 bytes. But subsequently, a revised value of 556 bytes was reported. Why did this change?

The function printf() was used from the ESP environment (lines 35 and 36). This is provided by the linked in newlib library (sourceware.org/newlib). The printf() call used an additional 1060 stack bytes to report the high water mark. So in the following iteration, the reduced high water mark was reported (line 33 tests for a change in the high water mark).

How large does the stack need to be in this program? It would appear that no less than 2048 – 556 = 1492 bytes are required. But you should always add some margin to that, perhaps 1800 bytes. On the other hand, if you don't need to keep that printf() call in the final code, the call could be removed and the stack size reduced. In that case, you might get away with approximately 2048 – 1616 = 432 bytes, plus a small margin.

Memory Management in FreeRTOS
The astute reader will recognize that the created task had at least one block of memory allocated from the heap for use as the stack (otherwise where did the stack come from?) FreeRTOS also uses a small Task Control Block (TCB), which is also allocated to manage the task's state etc.

Standard FreeRTOS provides five heap implementations for platform integrators. Espressif has provided their implementation because of the different types of memory that it must manage. Thus multiple heaps are managed according to type. For most application purposes, the standard malloc() and free() functions should be used. The types of memory managed on the ESP32 platform includes:

- Data RAM (Espressif calls DRAM), that is used to hold data. This is the normal memory heap.

- IRAM (Instruction RAM), used to hold executable code only. When accessing this storage as data, the access must be 32-bit aligned.

- D/IRAM is RAM that can be used as data or instruction memory.

- It is also possible to attach external SPI RAM on the ESP32.

ESP provides the function heap_caps_malloc() to permit the caller to choose the heap from which to allocate from:

```
void *heap_caps_malloc(size_t size,uint32_t caps)
void heap_caps_free(void *ptr)
```

The argument caps must be one of the ESP supported MALLOC_CAP_* macro values. Details about this and more are found in the ESP documentation.[3] The generic free() function can be used instead of heap_caps_free() if you prefer.

Static Tasks
Previously, we noted that the task's stack and task control block is dynamically allocated from the heap. This is a serious issue for safety-critical applications. What happens if your device needs to emergency power down your appliance and you can't create the task to perform this? Reasons for failure include memory exhaustion or fragmentation. As Dave Jones of the EEVBlog would say, "the magic smoke will escape!" [2]

One approach suitable for hobby projects is to create the task at application startup as usual and then suspend it until it is needed. But for life and death applications that demand the utmost in safety, the task is best created with pre-arranged (static) memory instead. Then, regardless of how fragmented or exhausted the heap becomes, the task is guaranteed to create successfully.

The functions that create tasks with statically allocated memory are:

- xTaskCreateStaticPinnedToCore() (Espressif only)
- xTaskCreateStatic()

Unfortunately for Arduino, this functionality is not configured into the Arduino (ESP) SDK. Advanced users with some effort can modify and rebuild their version of the SDK, but this is beyond the scope of this text. Those using the ESP-IDF can configure the functionality by performing a make menuconfig and then enable the option CONFIG_SUPPORT_STATIC_ALLOCATION.

Because this is an important topic, let's briefly examine what is involved in statically creating a task. Listing 2-3 shows a code fragment, which statically allocates a stack (line 1) and a Task Control Block (TCB, line 2).

Note: Students of C/C++ should be aware that the "static" keyword just means that these declaration symbols are local to the current file (compilation unit). Omitting keyword "static" would make the values *external* in scope and potentially clash with other globally defined symbols in modules that you are linking with. It is a best practice to declare global variables with the static keyword unless you intend those values to be shared with other linked modules.

In this example, we want a stack size of 2048 bytes. Since the StackType_t might be a word data type on some platforms, a calculation is used for the array size for portability.

```
0001: static StackType_t stack[2048/sizeof(StackType_t)];
0002: static StaticTask_t tcb;
0003:
0004:   TaskHandle_t taskh; // Task handle
0005:   ...
0006:   taskh = xTaskCreateStatic(
0007:     task_func,    // Function
0008:     "statictsk",  // name
0009:     2048,         // Stack size
0010:     &args,        // Pointer to args
0011:     1,            // Priority
0012:     &stack[0],    // The stack
0013:     &tcb          // TCB
0014:   );
```

Listing 2-3. Static task creation snippet.

With the stack and TCB declared (and thus allocated), the call to xTaskCreateStatic() is made. Pay careful attention to the fact that the task handle is the return value in this call. Argument 6 (line 12) passes a pointer to the start of the stack. The seventh argument (line 13) passes the address of the TCB to the task create function.

The only way this function can fail is by passing bad argument values. If either the stack or the TCB arguments are a nullptr for example, the create task will fail and return a nullptr for the task handle.

The API function creates your task by:

1. Initializing the TCB
2. Initializing the stack storage, so that stack usage can be determined.
3. Linking the task (TCB) into the scheduler task priority list.

If the created task's priority is higher than the creating task, the new task will start execution immediately. More will be said about task priorities in the chapter Task Priorities.

Task Delete
There are times when it is desirable to delete (terminate) a task. Reviewing Table 2-1, it was noted that there are gaps in the task numbering. These are due to tasks that were created and run at startup but have since completed and been deleted. Looking again at the demo task in Listing 2-2, the main loop in lines 70 to 72 does nothing except call delay(). Every time that the FreeRTOS scheduler runs the main task, loop() is called again, only to call delay() again. When delay() is invoked, the main task is put back to sleep for the requested time period, accomplishing nothing.

This is undesirable because:

1. Memory allocated to the main task is not well utilized (the stack is barely used).
2. Execution of the main task is a total waste of time.

A task may delete *another* task, or a task can delete itself. We could delete the main task from within the setup() function, or from the loop() function (these functions both run from the same loopTask). These would be an examples of deleting self. In the next example, the task delete will be performed in the loop() function. This is ok because once the task is deleted, the loop() code will no longer be invoked.

The program is illustrated in Listing 2-3, starting with line 24. The top portion of the code remains the same as Listing 2-2.

```
0024: static void led_task_func(void *argp) {
0025:   s_led *ledp = (s_led*)argp;
0026:   unsigned stack_hwm = 0, temp;
0027:
0028:   delay(1000);
0029:
0030:   for (;;) {
0031:     digitalWrite(ledp->gpio,ledp->state ^= 1);
0032:     temp = uxTaskGetStackHighWaterMark(nullptr);
0033:     if ( !stack_hwm || temp < stack_hwm ) {
0034:       stack_hwm = temp;
0035:       printf("Task for gpio %d has stack hwm %u, heap %u bytes\n",
0036:         ledp->gpio,stack_hwm,
0037:         unsigned(xPortGetFreeHeapSize()));
0038:     }
0039:     delay(ledp->napms);
0040:   }
0041: }
0042:
0043: void setup() {
0044:   int app_cpu = 0;    // CPU number
0045:
0046:   delay(500);         // Pause for serial setup
0047:
0048:   app_cpu = xPortGetCoreID();
0049:   printf("app_cpu is %d (%s core)\n",
0050:     app_cpu,
0051:     app_cpu > 0 ? "Dual" : "Single");
0052:
0053:   printf("LEDs on gpios: ");
0054:   for ( auto& led : leds ) {
0055:     pinMode(led.gpio,OUTPUT);
0056:     digitalWrite(led.gpio,LOW);
0057:     xTaskCreatePinnedToCore(
0058:       led_task_func,
0059:       "led_task",
0060:       2048,
0061:       &led,
0062:       1,
0063:       &led.taskh,
0064:       app_cpu
0065:     );
0066:     printf("%d ",led.gpio);
0067:   }
0068:   putchar('\n');
0069:   printf("There are %u heap bytes available.\n",
0070:     unsigned(xPortGetFreeHeapSize()));
0071: }
0072:
0073: void loop() {
0074:   // Delete self (main task)
0075:   vTaskDelete(nullptr);
0076: }
```

The call to vTaskDelete() occurs in line 75 of the listing. Notice that a nullptr was passed as the task handle to imply "self". To prove that the task was deleted, the setup() routine uses the FreeRTOS function xPortGetFreeHeapSize() in line 69 to report the number of heap bytes available. The LED driving task was also modified to report the number of heap bytes available (lines 35 to 37). Now let’s run the demonstration and examine the serial monitor output:

```
app_cpu is 1 (Dual core)
LEDs on gpios: 12 13 15
There are 281992 heap bytes available.
Task for gpio 15 has stack hwm 1624, heap 290564 bytes
Task for gpio 13 has stack hwm 1616, heap 290564 bytes
Task for gpio 12 has stack hwm 1620, heap 290564 bytes
Task for gpio 13 has stack hwm 556, heap 290564 bytes
Task for gpio 12 has stack hwm 500, heap 290564 bytes
Task for gpio 15 has stack hwm 564, heap 290564 bytes
```

Initially, setup() reports that there are 281,992 bytes available after creating the three LED driver threads. But after the loop() function runs and deletes itself, we see that the LED driving tasks later report that 290,564 bytes are available. This tells us that 8,572 bytes were freed when the loopTask was deleted. The blinking of the LEDs proves that the three created tasks continue to run ok.

This is getting ahead of ourselves because this topic will be covered later. But a fine point needs to be mentioned at this point. The vTaskDelete() function will terminate the execution of the specified task immediately, as you'd expect. However, when the memory is released depends upon who the caller is.

When a task deletes another, other than self, the task is immediately terminated *and* the memory *immediately* released. When a task deletes self, the task is immediately terminated but the memory is not immediately released. The memory cleanup is left until the IDLE task executes. That is why line 28 delays for one second in the task function led_task_func. This delay allows the IDLE task to schedule, necessary because the IDLE task runs at priority zero (lowest priority).

If you didn't care about the reclaimed memory, you can ignore the issue and be content that the deleted task is no longer executing. Alternatively, you could arrange *another* task to perform the task delete, and then the task termination and memory release would be immediate.

Finally, the task must only be deleted once. It is a fatal error to delete a task that has already been deleted, much like freeing already freed storage.

Note: You can also call vTaskDelete() on a static task. No memory is released in this scenario, but it does modify the static task's TCB so that it will not get scheduled again. In other words, the static task is terminated with no memory reclaim.

Task Suspend/Resume

Another option in task control is to choose when it should be scheduled. FreeRTOS permits the caller to suspend and resume tasks. Listing 2-4 illustrates the loop() code that is used in the task_suspend.ino program (the remainder of the code is the same as Listing 2-3). Instead of deleting the main task, we use the main loop() function to suspend the middle LED driving task for five seconds, and then resume it for five more, and repeat.

```
void loop() {
  delay(5000);

  printf("Suspending middle LED task.\n");
  vTaskSuspend(leds[1].taskh);
  delay(5000);
  printf("Resuming middle LED task.\n");
  vTaskResume(leds[1].taskh);
}
```

Listing 2-4. The task_suspend/task_suspend.ino program.

This results in the following serial monitor output:

```
app_cpu is 1 (Dual core)
LEDs on gpios: 12 13 15
There are 281992 heap bytes available.
Task for gpio 15 has stack hwm 1624, heap 281992 bytes
Task for gpio 13 has stack hwm 1616, heap 281992 bytes
Task for gpio 12 has stack hwm 1620, heap 281992 bytes
Task for gpio 13 has stack hwm 556, heap 281992 bytes
Task for gpio 12 has stack hwm 500, heap 281992 bytes
Task for gpio 15 has stack hwm 564, heap 281992 bytes
Suspending middle LED task.
Resuming middle LED task.
Suspending middle LED task.
Resuming middle LED task.
Suspending middle LED task.
Resuming middle LED task.
```

No memory is saved by suspending a task because the task still exists. The scheduler is simply informed not to give it any CPU time. After the message "Suspending middle LED task", you will see that the middle LED stops flashing (driven by GPIO 13). Due to timing, it may just happen to stay lit during suspension. Five seconds later, the task is resumed again and the middle LED flashes again. There other options for synchronizing tasks, including those covered in chapter "Events".

While this API demonstrates suspension and resumption, it may not always be desirable to do this to a task. A suspended task may be left in the middle of an important multi-step function. Or it may hold a lock that another task needs to obtain. For these reasons, other FreeRTOS synchronization functions may be preferred.

Finding Yourself
Sometimes the same function's code is used for multiple tasks, like the demonstration program in Listing 2-2. The same function led_task_func() was used to drive three different LEDs, from different tasks. These particular tasks had access to their task handle by using the passed in argument (line 25 of Listing 2-2), and access to ledp->taskh. But how could it obtain a task handle, if the handle was not provided?

FreeRTOS provides a convenience function named xTaskGetCurrentTaskHandle() for this. It takes no arguments and provides the current task's handle as the return value:

```
TaskHandle_t taskh = xTaskGetCurrentTaHandle();
```

Task Time Slice
Earlier we discussed how the FreeRTOS scheduler gives each task a time slice. The question that naturally follows is how long is this time slice? Or framed another way, how many different tasks can execute concurrently in a given second? Is this something that can be

measured? Indeed we can!

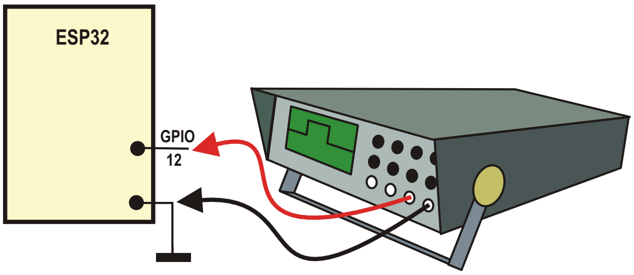


Figure 2-2. Scoping GPIO12.

A program with two competing tasks – one writing a high to a GPIO pin, and the other writing a low to the same GPIO, on the same CPU can measure the time slice period. This should show a pulse on an oscilloscope (Figure 2-2) as preemptive context changes occur.

On the ESP32 we have the advantage of having a whole CPU (1) to ourselves because most of the housekeeping runs on CPU 0. Referring to Table 2-1 again, the column labeled CPU identifies tasks that run on CPU 1. Included in that list is our venerable "loopTask", "IDLE1" and "ipc1". IDLE1 runs at the lowest priority zero, so it will not interfere with our priority level 1 tasks (see chapter Priority). A potentially interfering task is the "ipc1" task, which has a high priority of 24. But due to it being blocked waiting for an event, it'll not likely interfere.

Our demonstration program in Listing 2-5, creates two tasks named gpio_on and gpio_off, at priority 1. We'll also delete the loopTask, so that it no longer executes. So long as the high priority task "ipc1" doesn't interfere, it is expected that only the gpio_on and gpio_off tasks will execute on CPU 1. This should demonstrate that task gpio_on is executing when the GPIO is high, and another period of low when task gpio_off executes.

```
// ticks.ino
// MIT License (see file LICENSE)

#define GPIO  12

static void gpio_on(void *argp) {
  for (;;) {
    digitalWrite(GPIO,HIGH);
  }
}

static void gpio_off(void *argp) {
  for (;;) {
    digitalWrite(GPIO,LOW);
  }
}

void setup() {
  int app_cpu = xPortGetCoreID();

  pinMode(GPIO,OUTPUT);
  delay(1000);
  printf("Setup started..\n");

  xTaskCreatePinnedToCore(
    gpio_on,
    "gpio_on",
    2048,
    nullptr,
    1,
    nullptr,
    app_cpu
  );
  xTaskCreatePinnedToCore(
    gpio_off,
    "gpio_off",
    2048,
    nullptr,
    1,
    nullptr,
    app_cpu
  );
}

void loop() {
  vTaskDelete(xTaskGetCurrentTaskHandle());
}
```

Listing 2-5. Program ticks/tick.ino to measure the ESP32 time slice.

Figure 2-3 shows a scope capture of the GPIO 12 signal when the program ran. The horizontal divisions are 500 µsec apart, measuring a one millisecond pulse width. The low between the pulses is also one millisecond apart, demonstrating that our two tasks were getting all the CPU. From this, we can conclude that the task preemption occurs at one millisecond intervals. This permits a minimum of 1000 task slices to execute per second. More slices are possible if a given task doesn't use the full slice. Hold that thought.

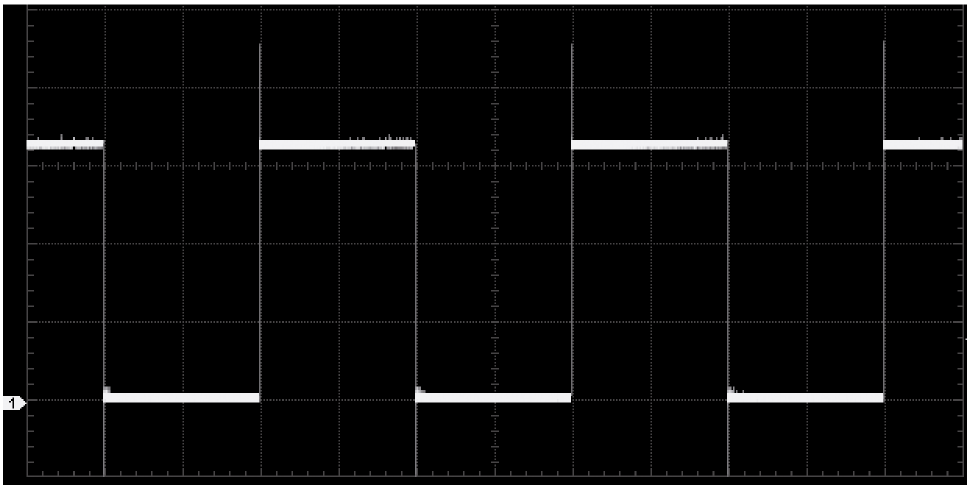


Figure 2-3. Captured scope trace of GPIO 12,
with horizontal 500 usec / division, vertical 1 volt / division.

While the ESP8266 Arduino IDE does not support the FreeRTOS API, a similar program was run, compiled with the ESP-IDF. For the curious, that experiment measured a 10 ms pulse width instead (100 task slices per second).

This tick period is easily reconfigurable from the ESP-IDF for the ESP32 and ESP8266. But the Arduino environment is a prebuilt system and so this parameter remains fixed. The source code is available for modification but is not recommended for beginners.

Yielding CPU
It was shown that the Arduino ESP32 time slice is one millisecond. But what if your task doesn't use that full slice because it is waiting for an event? Listing 2-6 shows a slightly modified version of the previous program, to discover this. The only material change is lines 8 through 10. The for loop in line 8, limits the writing of the GPIO to 1000 iterations. Then the FreeRTOS function taskYIELD() is called (this is a macro that makes the actual call on your behalf).

FreeRTOS taskYIELD() tells the task scheduler that you have nothing further to do and that you want to hand over the CPU to another worthy task. Instead of spinning in the loop, this allows something else useful to be done. The scheduler will pass control to another task of the *same* priority. Since only the gpio_on task was modified to yield, the gpio_off task should continue to use its entire time slice as before. Run it and check the result on the scope (Figure 2-4).

```
0006: static void gpio_on(void *argp) {
0007:   for (;;) {
0008:     for ( short x=0; x<1000; ++x )
0009:       digitalWrite(GPIO,HIGH);
0010:     taskYIELD();
0011:   }
0012: }
0013:
0014: static void gpio_off(void *argp) {
0015:   for (;;) {
0016:     digitalWrite(GPIO,LOW);
0017:   }
0018: }
```

Listing 2-6. The change to function gpio_on in the task_yield/task_yield.ino program.

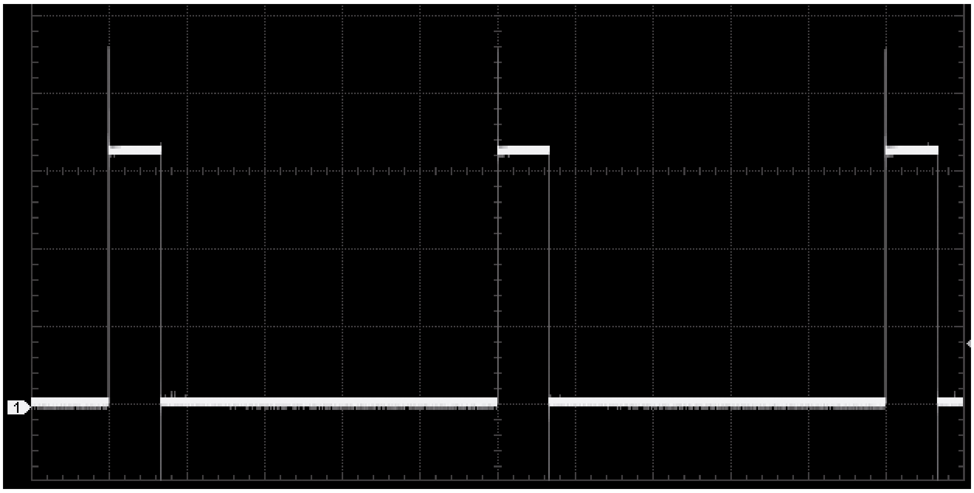


Figure 2-4. Scope trace from Listing 2-6.
Horizontal is 200µsec / division, vertical is 1 volt / division.

The high time is from the execution of task gpio_on (Listing 2-6 in lines 6 to 12). What is interesting is that the time from the falling edge to the next rising edge is less than one millisecond. So what happened to task gpio_off's time slice?

If we measure from leading edge to leading edge of the scope trace, it measures one millisecond. As expected, the short high pulse indicates that the gpio_on task ran but later gave up its time slice when taskYIELD() was called. The FreeRTOS scheduler then gave the remainder of that time slice to the gpio_off task until the next timer "tick".

What can we conclude? The full time slice is not guaranteed. Instead, taskYIELD() call causes the FreeRTOS scheduler to share the CPU with another task, but only until the timer expires (the "tick"). A tick occurs on the ESP32 at one millisecond intervals. The scheduler will run another task at the same priority as the current one (else the current task would not be executed if a higher priority task was ready). Tasks at the same priority share the CPU in a "round-robin" fashion, to distribute access of the CPU.

Figure 2-5 illustrates the scheduling that occurs at program startup and into the first round of round-robin. Beginning with the setup() and loop() functions the "loopTask" runs until the task deletes itself. After that, the two created tasks gpio_on and gpio_off begin to schedule. Task gpio_on was the first to be created, so it schedules first. It executes until it calls taskYIELD(). This causes a *voluntary* context switch at the dotted line. The scheduler then runs task gpio_off until the preemption timer tick occurs, which marks the end of the time slice. In round-robin fashion, task gpio_on resumes again and runs until the task-YIELD() call, and around we go.

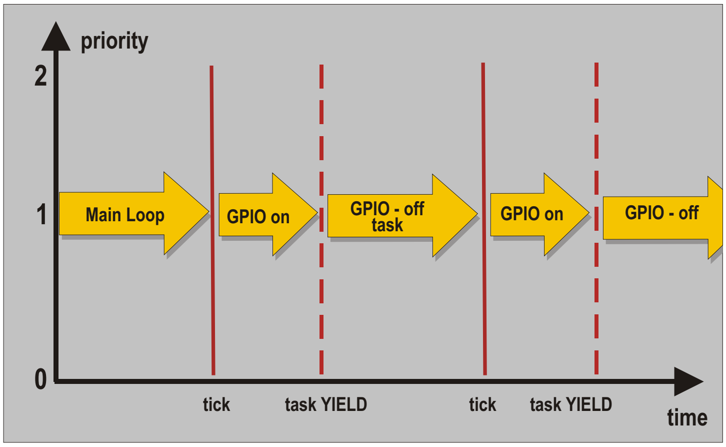


Figure 2-5. FreeRTOS scheduling of the program in Listing 2-6, using taskYIELD().

The round-robin scheduling has a sneaky side-effect for the unwary. If your executing task is the only task *ready* to run on a given CPU, at a given priority, then the scheduler will return immediately from your call to taskYIELD() and resume execution. This happens because it cannot give the CPU to any other task. In FreeRTOS, lower priority tasks cannot execute while a higher priority task is ready to run.

Assert Macro

The programs presented so far have little or no error checking. This makes the code easier to read but potentially leads to painful debugging sessions. A good program, especially a finished product, should include checks at various points of failure. A prime candidate would be checking the result of the function xTaskCreatePinnedToCore(). In the POSIX world (Unix/Linux/MacOS/*BSD systems), a popular facility for this purpose is the C assert macro (provided by including assert.h). The ESP32 Arduino environment already includes this facility, so all you have to do is to apply it.

For example, the snippet below creates the gpio_on task, and deposits the handle of the task into variable h1, if it succeeds. If we add the simple macro call in line 34, we can check the handle, and be assured that the error would be caught if it failed for any reason. In other words, if the expression h1 != nullptr is true, then all is well and nothing happens. But if the handle is null (the expression evaluates false), then the error is reported to the serial monitor, and the program aborts.

The worst type of error is the unreported kind. Unreported errors lead to bad program behaviour, which then leads to guesswork.

```
TaskHandle_t h1;
...
 xTaskCreatePinnedToCore(
   gpio_on,
   "gpio_on",
   2048,
   nullptr,
   1,
   &h1,     // Task handle
   app_cpu
 );
assert(h1 != nullptr);
```

The following assert macro call was coded in a test setup() function, to demonstrate a caught assertion error:

```
  assert(1==0); // Never true that 1 == 0
```

When this program is flashed and booted, the serial monitor produces the following messages:

```
Rebooting...
ets Jun  8 2016 00:22:57

rst:0xc (SW_CPU_RESET),boot:0x17 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
assertion "1==0" failed: file "test.ino", line 3, function: void setup()
abort() was called at PC 0x400d6307 on core 1

Backtrace: 0x4008af94:0x3ffb1f20 0x4008b1c1:0x3ffb1f40 0x400d6307:0x3ffb1f60
0x400d0b26:0x3ffb1f90 0x400e653b:0x3ffb1fb0 0x40087cbd:0x3ffb1fd0
```

The most important information to you as the programmer is that the failure occurred in line 3 of the file "test.ino" in the function setup(). This points to the fact that an error occurred, and where. The macro call is low overhead and can save you a weekend of debugging.

Summary

FreeRTOS makes scheduling separate tasks easy to manage for the application designer. Tasks can be dynamically or statically allocated as the need dictates. Task preemption and voluntary context switches have been described and tested. The API for task suspension and resumption has been reviewed, along with task deletion. At this point, you should have a good mastery of task control.

Armed with tasks, the designer will quickly discover that he/she needs to communicate and synchronize with other tasks within the system. How is this reliably performed? A large portion of the FreeRTOS API is dedicated to precisely this. The next chapter explores the queue API, which provides one solution in this problem space.

Exercises

1. How do you obtain the currently executing task handle?
2. How do you give up the CPU to another task?
3. Which CPU core do application programs execute on for the ESP32?
4. What is the name of the default task provided by the Arduino environment?
5. What FreeRTOS function is used to suspend one task?
6. Can a task delete itself and if so how?
7. When is the task's stack released when a task deletes itself? Immediately or later during the IDLE task?
8. What causes the preemption in FreeRTOS for ESP32?
9. How often do the ESP32 tick interrupts occur?

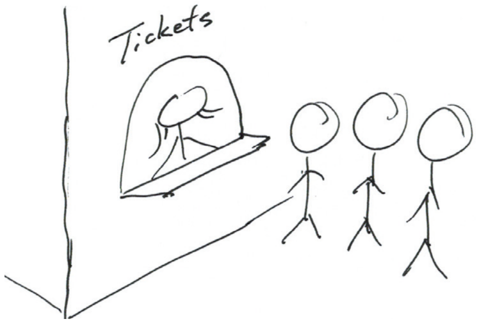
Web Resources

[1] "ESP32." Wikipedia. Wikimedia Foundation, December 16, 2019.
https://en.wikipedia.org/wiki/ESP32.

[2] EEVblog. "EEVblog." YouTube. December 1, 2019.
https://www.youtube.com/channel/UC2DjFE7Xf11URZqWBigcVOQ

[3] "Heap Memory Allocation." ESP. December 1, 2019.
https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/system/mem_alloc.html.

Chapter 3 • Queues



Queue up for some fun.

When your code calls a function, data items are passed as arguments. The timing of the exchange is simple because the calling code is suspended while the called function processes the arguments and returns a value. All of this occurs as a *synchronous* event. But how do two independently executing tasks pass data items safely between them?

One FreeRTOS solution to this problem is the queue. Because tasks execute independently, the queue provides a safe buffer between the producing tasks (*producers*) and the consuming tasks (*consumers*). Let's learn how to exploit this facility.

Queue Characteristics

Everyone has had experiences with queues in daily life. When you enter a bank or line up for concert tickets, you have been in a queue. Maybe you chose the shortest lineup at the supermarket checkout, or got a numbered ticket that is called out when you could be served. All these are examples of queues.

Queuing theory is often the subject of serious studies and papers because the topic is so important. Here, we'll just focus on the practical aspects of FreeRTOS queues after introducing a few concepts.

Arrival Pattern

Items entering a queue can arrive in different patterns. This has a bearing on how you choose to use the FreeRTOS queue. Consider the following patterns:

- **Balking:** can a data item sometimes not queue due to some consideration?

- **Reneging:** can a data item cancel its membership in the queue?

A common consideration for FreeRTOS queues is what action to take when it becomes full. Should the code:

1. Block until space becomes available to queue the data?
2. Give up and try something else.

In point 1, the calling task is suspended (blocked) until the data item is successfully queued. For that to succeed for a full queue, one or more items have to be removed by a receiver. What if the queuing occurs within an ISR (Interrupt Service Routine)? Blocking is not an option there, so the action to give up and do something else (balking) may be more appropriate.

When you wait a long time in a lineup, you might choose to leave a queue. This is an example of reneging. FreeRTOS does not support this queue feature. Once a data item is queued, it remains there until it is received or the queue contents are discarded.

Capacity
The capacity of a queue is another consideration. There are two factors:

- Fixed capacity.
- Unlimited capacity.

You set the capacity of the queue when you create it in FreeRTOS. The fixed capacity is specified in terms of the maximum number of items it may hold (also known as the queue depth). FreeRTOS does *not* support unlimited or varying queue capacities. Only fixed-length queues can be created.

The fact that your queue is fixed in capacity, has consequences for *flow control*. When a queue is full, no more items can be added. So what does your code do when the queue becomes full? Will the queuing task block until the item can be queued? Or will it take some other action?

Service Discipline
There are different service disciplines that a queue may support. A priority-based queue, for example, may receive high priority data items ahead of low priority items. The discipline that we are most familiar with is the FIFO queue (First In First Out). If you want to see people get angry, just violate the FIFO rule in a long lineup.

FreeRTOS implements the FIFO queue (with one exception). The first data item queued will be the first data item received. The exception is that FreeRTOS permits you to choose to push to either end of the queue (*front or back*). Pushing a data item to the front has the effect of making that data item first in priority. But this is still not a true priority queue. If there was already an unreceived first priority item in that queue, the new item has just become the new first priority item. True priority queues maintain FIFO order within a given priority.

Sources and Destinations
The FreeRTOS queue can be used with single or multiple queuing sources. A single task may queue items, or several tasks may simultaneously be queuing data items. FreeRTOS ensures that the queuing operation is atomic. This is a vital aspect of FreeRTOS when several tasks are executing. It is impossible to partially push a data item.

In the same manner, there can be one or more tasks receiving items from a given queue. Multiple receivers are useful when you want to share the burden of processing. On the dual-core ESP32 for example, a task running on each CPU permits up to two tasks to simultaneously process queued items. Otherwise concurrent receivers can process queued items.

Basic Queue API
Let's now examine the basic queue API that FreeRTOS supports. The next few sections will indicate how queues are created, data items added and received, etc. Seeing the moving parts will take the mystery out of it.

Creating Static Queues
Like static tasks, the Arduino environment does not support the creation of static queues. But it is instructive to examine this FreeRTOS capability first because it clearly defines the involved data components. The following are the data elements:

```
typedef unsigned long qitem_t;  // Some arbitrary data item type
#define QUEUE_DEPTH 10          // Maximum queue depth

static uint8_t qstorage[QUEUE_DEPTH * sizeof(qitem_t)];
static StaticQueue_t qobj;     // Queue object

QueueHandle_t qh;              // Queue handle
```

It may seem as if there is a lot to digest, so let's break it down:

1. Almost any data item can be sent and received through the FreeRTOS queue, so I've represented the simple data item with a typedef and gave it the name of qitem_t. The type can be any type, perhaps a struct, a float, a short integer, character, bool, or whatever your application needs. You are not required to use a typedef.

2. A queue must have a fixed maximum depth under FreeRTOS. Somewhere you have to specify the depth of the queue. Here it was done using a macro named QUEUE_DEPTH for illustration purposes. You don't have to use a macro in your code, but you may find it useful.

3. Storage must be allocated to hold the queued data items. This is defined in the uint8_t array named qstorage. Notice that the storage size is computed in bytes, which is the number of data items (QUEUE_DEPTH) times the size of each item (sizeof(qitem_t)).

4. There is also the queue *object* itself. The object manages the state of the queue, indexes into the queue storage, and provides priority-based ordering for tasks.

5. Once the queue is created, a handle is returned (qh).

6. With these four elements and the returned handle, you have enough to create and work with a queue without using dynamic memory. When static queue allocation is supported, the following FreeRTOS call is used:

```
qh = xQueueCreateStatic(  // Handle is returned
  QUEUE_DEPTH,            // UBaseType_t (Queue depth)
  sizeof(qitem_t),        // UBaseType_t (byte size of each item)
  &qstorage[0],           // Start of queue storage
  &qobj);                 // Address of queue object
```

Given that this is a static creation, the only way this call can fail is if you provide inappropriate argument values. If the handle returned is nullptr/NULL, then the create failed. This is a good thing to check with the assert() macro.

Note: The queue storage space must be the queue depth times the size of each item queued. If the storage provided is smaller than this amount, memory corruption will occur with the subsequent use of the queue.

From the example shown, the maximum queue depth is provided in the first parameter. Parameter 2 indicates the size of each queued item. The third parameter is the starting address of the queue storage area for the items. The last argument points to the queue object, which will maintain the queue state.

Queuing an Item

Continuing with the previous data types, let's add one data item to the back of the queue.

```
q_item_t my_item = 42;      // A data value to be queued
TickType_t wait_ticks = 2;  // How many ticks to wait

BaseType_t rc = xQueueSendToBack(
  qh,               // Queue handle
  &my_item,         // Item value to queue
  wait_ticks);      // How long to wait when full
```

The queue handle (qh) in the first argument, indicates which queue to add the data item to (you may have several queues). The second argument is a pointer to the data item that you want to add to the queue. The final argument indicates what action should be taken if the queue is full. When supplied as zero, the call immediately fails when the queue is full and the returned code (rc) will be assigned the value errQUEUE_FULL. When the item is successfully added to the queue, the value pdPASS is returned instead.

When the argument wait_ticks is greater than zero, the calling task is suspended when the queue is full. The task will remain suspended until the queue is no longer full and an item can be added. The task is said to be *blocked* under these conditions. The time that the task will be blocked is specified as *ticks*. If you would prefer to specify a time in milliseconds instead, use the macro pdMS_TO_TICKS(ms) instead. The special macro value portMAX_DE-

LAY can be used to have the task block forever when the queue is full.

Note: The item size must match the item size provided in the original queue creation. The item size is fixed (as bytes). The second argument to xQueueSendToBack () only supplies the starting address of the data item. Therefore the item size implied is the size stored within the queue object, established by the xQueueStaticCreate() call. A size mismatch will cause data integrity issues.

There is also the capability to add to the front of the queue using the function xQueueSendToFront(). For backward compatibility, there is also xQueueSend(), which is equivalent to xQueueSendToBack(), but that name should not be used in new code.

The data item added to the queue requires a physical memory copy of the item storage. For this reason, you should avoid large objects for efficiency. Because the data item is *copied,* the queuing code does not need to be concerned about data item lifetime.

Receiving from a Queue

Receiving data items from a queue is just as easy as adding items. The first argument to xQueueReceive is the queue handle to receive from. The second parameter is a pointer to the data item storage to be received into, while the final argument is wait_ticks. This tick time specifies how long to block the calling task when the queue is empty. When zero is provided then the call immediately fails when the queue is empty (rc = errQUEUE_EMPTY). Otherwise, the task is blocked for up to the specified number of ticks or until data arrives. When a data item is successfully received, the returned code is pdPASS.

```
qitem_t item;  // Received data item
TickType wait_ticks = 12;

BaseType_t rc = xQueueReceive(
  qh,          // Queue handle
  &item,       // Pointer to data item
  wait_ticks); // Time to wait
```

- rc == pdPASS when a data item has been returned.
- rc == pdQUEUE_EMPTY when no data item has been returned (timeout)

Note: The receiving data item storage must match the size provided in the queue create call. The item size is stored within the QueueType_t object. If the supplied storage is smaller than expected, memory corruption will occur.

The received data item is received using a memory copy. Receiving always occurs from the front of the (FIFO) queue. There is no function for receiving from the rear.

Dynamic Queue Creation

Creating a dynamically allocated queue is much easier than the static queue since FreeRTOS allocates all of the necessary storage for you. This is the function that the ESP32 Arduino user will use:

```
QueueHandle qh = xQueueCreate(
  QUEUE_DEPTH,       // UBaseType_t (max number of items)
  sizeof(qitem_t)); // UBaseType_t (size of each data item)
```

It is simply necessary to specify the queue depth in the first argument and the size of each data item in the second argument. The queue storage and the QueueType_t object are both allocated from the heap by FreeRTOS. The risk is that this might fail if your heap is exhausted or overly fragmented. You should always check that the returned handle (qh) is not nullptr/NULL.

Queue Delete

If you don't need a queue at some point within your application, it can be deleted (with preconditions). Simply supply the queue handle to xQueueDelete():

```
xQueueDelete(qh);   // Delete queue by handle
```

There is no return value from the call. A static queue may also be deleted (this disables any task interaction with the queue).

Note: It is invalid to delete an already deleted queue. A good practice is to null the handle of a resource that has been deleted. Then the code can test if it was already destroyed.

FreeRTOS requires that a queue never be deleted if there are tasks blocked waiting for a queue (task blocking can occur on full or empty queues). If there is any doubt about this, it is best to *not* delete the queue.

Queue Reset

Applications may sometimes need to reset (empty) a queue. Any items held in a reset queue are simply discarded. Simply pass the queue handle to the xQueueReset() function:

```
xQueueReset(qh); // Reset the queue by handle
```

This call always succeeds provided that the queue handle is valid. So there is no need to check the return value (the return value is present for backward compatibility only).

Note: Resetting a queue does not guarantee that there will be no tasks left blocking on the queue afterwards. A task calling a xQueueReceive() can block on an empty queue. Consequently, this does not guarantee safety before a queue delete.

Task Scheduling
In the previous chapter, the concept of FreeRTOS task scheduling was introduced. The highest priority task executes unless its execution is *blocked* by some operation. When adding or receiving entries from queues, there are conditions that can lead to *blocking* the calling task. The nature of this depends upon the operation and its calling parameters. Let's now spend a moment expanding on some subtle aspects of queue operation.

Blocked while Adding
When adding an item to a queue, the calling task can be *blocked* when the queue is *full*. How this is handled depends upon the third argument (wait_ticks):

- When wait_ticks is zero, no waiting is performed and the queue add immediately fails when the queue is full. The calling task *never* blocks in this case.

- When wait_ticks is greater than zero and the queue is full, the calling task will *block* for the specified number of ticks. Otherwise, when there is room, the add succeeds and the call returns immediately.

- When wait_ticks carries the value portMAX_DELAY and the queue is full, the calling task *blocks* forever when there is no space. When space becomes available, the call returns successfully.

When the queue is full, what exactly happens when the calling task becomes blocked? It gets changed to the *not ready* state. If there are one or more *other* tasks at the same priority, they are scheduled to run next instead of returning from the present call. After all, the current call cannot succeed as long as the queue remains full. If there are no equal priority tasks, then a lower priority task is chosen.

Sometimes an application cannot tolerate waiting, which is the purpose of the wait_ticks argument. When waiting is not an option, the value zero can be specified for the wait_ticks argument. Then the program must take corrective action when an error is returned. Other applications may be able to tolerate waiting for a time. If that time elapses without success however, the application must still be prepared to handle the error code returned. Finally, if success is the only acceptable option, then waiting forever with portMAX_DELAY is the correct choice.

A task that is blocked on a full queue becomes *unblocked* when the queue regains space for adding another item. The task returns to the ready state. When multiple tasks are attempting to add to the same full queue, the highest priority task will become unblocked first and add the item.

Blocked while Receiving
Opposite to sending, there is receiving. When the queue is empty, the calling task cannot succeed because there is no item to return. Consequently, an application must decide what to do for an empty queue:

- Supply zero for wait_ticks and handle the error returned when the queue is empty.

- Supply a greater than zero wait_ticks value and handle the error returned if the call returns without receiving a data item.

- Supply portMAX_DELAY and return only when a data item is received.

When the receiving task becomes blocked, then another task with equal or lower priority will be scheduled to run next.

A task blocked on an empty queue will become unblocked when an item is added to the queue. When multiple tasks are receiving from the same queue, the highest priority task will be the task that is unblocked and receive the item.

Demonstration
Some automobiles have a traction control button on the dashboard. By default, the traction control is enabled when you start the vehicle (and the LED remains off). The idea is that if the computer detects a difference in the spinning of wheels, it assumes an unsafe driving condition is developing and disables the ignition and fuel going into the engine. This rapidly reduces power to keep the driver from going out of control.

But if you get stuck in a snowed-in parking lot, you don't want the engine cutting out, just when you almost get out of the rut! The button allows the driver to disable that traction control by pressing the button. Confirmation that the traction control is disabled is provided by the illuminated LED.

This demonstration emulates the traction control system but illuminating the LED while it is in effect, instead (this simplifies gpio testing at program startup). The Arduino code de-bounces the input button signal and then queues an event to toggle the LED. The program is divided into two tasks:

1. Function debounce_task() is responsible for reading the input button and queu-ing a clean on/off signal.

2. Function led_task() is responsible for reading the queue for button events. The LED state is toggled if the received event is a push button "on" event. Button release events are ignored.

The demonstration starts with the LED on to make it easier to test the flashed program. If the LED fails to light when powered up and reset, then recheck your wiring. Otherwise, the principle is the same-- each button press will toggle the LED on and off.

Program Setup
Listing 3-1 illustrates the program found in file debounce.ino. Lines 5 and 6 define the GPIOs used for the LED and the input push button. The push button input GPIO is configured by the program to use an internal pullup resistor so that the input does not float. Don't use GPIOs 34 or higher for button inputs unless you add a 10k pullup resistor. The code is written so that a high on GPIO_LED turns the LED on (active high). Figure 3-1 illustrates the wiring used.

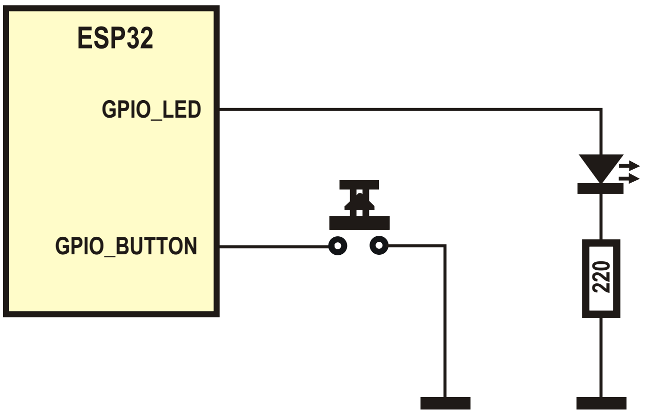


Figure 3-1. Wiring for ESP32 program in Listing 3-1.

The queue and the tasks are created in the setup() function (lines 62 to 94). Line 63 determines the current CPU for portability. The delay() in line 66 is useful if you have printf() statements, during debugging. This gives time for the Serial monitor to connect with the USB serial chip in "dev kit" ESP32 boards.

The queue is created in line 67. There are only two parameters, and the handle is returned:

1. 40 – The maximum number of items that can be queued.
2. sizeof(bool) – The size of each data item in bytes. This evaluates to 1.

The value 40 was chosen for the queue depth for this demonstration. This is overkill but does no harm since there is plenty of SRAM available. The data item being sent is a type bool, which is just a single byte with a 1 or 0 in it. The value 1 represents a debounced button press, while 0 represents a release.

The remainder of setup() creates the tasks, and the loop() function deletes the loopTask, since it is not used.

```
0001: // debounce.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // LED is active high
0005: #define GPIO_LED    12
```

```
#define GPIO_BUTTON 25

static QueueHandle_t queue;

//
// Button Debouncing task:
//
static void debounce_task(void *argp) {
  uint32_t level, state = 0, last = 0xFFFFFFFF;
  uint32_t mask = 0x7FFFFFFF;
  bool event;

  for (;;) {
    level = !!digitalRead(GPIO_BUTTON);
    state = (state << 1) | level;
    if ( (state & mask) == mask
      || (state & mask) == 0 ) {
      if ( level != last ) {
        event = !!level;
        if ( xQueueSendToBack(queue,&event,1) == pdPASS )
          last = level;
      }
    }
    taskYIELD();
  }
}

//
// LED queue receiving task
//
static void led_task(void *argp) {
  BaseType_t s;
  bool event, led = false;

  // Light LED initially
  digitalWrite(GPIO_LED,led);

  for (;;) {
    s = xQueueReceive(
      queue,
      &event,
      portMAX_DELAY
    );
    assert(s == pdPASS);
    if ( event ) {
      // Button press:
```

```
      // Toggle LED
      led ^= true;
      digitalWrite(GPIO_LED,led);
    }
  }
}

//
// Initialization:
//
void setup() {
  int app_cpu = xPortGetCoreID();
  TaskHandle_t h;
  BaseType_t rc;

  delay(2000);          // Allow USB to connect
  queue = xQueueCreate(40,sizeof(bool));
  assert(queue);

  pinMode(GPIO_LED,OUTPUT);
  pinMode(GPIO_BUTTON,INPUT_PULLUP);

  rc = xTaskCreatePinnedToCore(
    debounce_task,
    "debounce",
    2048,     // Stack size
    nullptr,  // No args
    1,        // Priority
    &h,       // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
  assert(h);

  rc = xTaskCreatePinnedToCore(
    led_task,
    "led",
    2048,     // Stack size
    nullptr,  // Not used
    1,        // Priority
    &h,       // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
  assert(h);
}
```

```
0098:
0099: // Not used:
0100: void loop() {
0101:   vTaskDelete(nullptr);
0102: }
```

Listing 3-1. Program listing of debounce/debounce.ino.

Debounce Task
The debounce task is found in lines 13 to 31. An infinite loop starts in line 18, which continually reads from the push button input (line 19). The result is guaranteed to be a simple 1 or 0, because of the !! C operator used. This value is saved in the variable named level. The variable named state is shifted left by one bit and the new signal level is or-ed into it. In this manner, each bit represents the push button state over time. This state is then anded with value mask. If after this, all bits match zero then the push button has stabilized into the pressed state (0). If the result is all one bits, then the button has stabilized into the released state (1).

Line 23 checks to see if this value different from the last known value. If it is the same as before, we simply ignore the event. Otherwise, we save variable event in line 24. The event is then queued in line 25, using the queue handle named queue, sending the boolean value (event), and use a timeout of 1 tick (third argument). If the queue is full, this call will block for up to 1 tick (1 millisecond on ESP32). If the queue should remain full, the value errQUEUE_FULL is returned instead (line 25). Upon success, we update variable named last to match the item just queued (line 26).

Line 29 calls taskYIELD() to share the CPU with our led_task() that is also running. Otherwise, the loop keeps reading the push button's GPIO pin looking for changes.

Bouncing metal contacts can create thousands of on/off electrical events on a GPIO pin. Here we sample frequently, gathering up to 31 bits worth of history. When the button is pressed and the contacts have settled, we will read 31 bits of consecutive zeros. If the contacts were still bouncing, there would be some 1 bits present. Conversely, upon release of the contacts, all bits will be 1, once the bouncing has ended.

LED Task
The LED task performs the receiving from the queue using the forever loop starting in line 43. The third argument uses the value portMAX_DELAY so that the xQueueReceive() function will not return until a data item has been returned. Once a value is received, it is checked to see if it is a button press event in line 50. When it is a button press event, the state of the LED is toggled in line 53 and the LED is driven in line 54.

Press Demonstration
This second demonstration is a simulation of a factory hydraulic press controller. A factory press providing tons of hydraulic pressure requires safety for the operator. This controller requires that the operator simultaneously press a button on the left and the right of the

console so that neither hand can be in the path of the press. If there is even a hint of a button release, the press immediately deactivates. The circuit for the demonstration is provided in Figure 3-2.

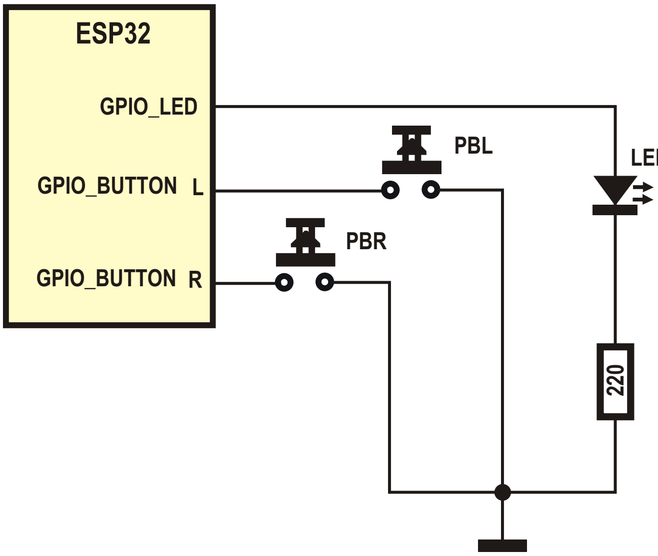


Figure 3-2. Circuit for the Hydraulic Press Demonstration.

The debouncing of the button differs somewhat from the earlier example. Here we must debounce the activation (button press), but any hint of a button *release* will trigger an immediate deactivation of the press. Button activation is debounced, but the release is not so that the release will be immediate.

Another way that this example differs from the first is that two tasks are queuing event data to one receiving task (see Listing 3-2). The left and right buttons each have their own task, receiving and debouncing one button input. When a debounced button press is detected, a positive GPIO number is sent as the event. Otherwise, a button release is sent as a negative GPIO number instead. This permits the receiving task to identify the source of the event.

```
// press.ino
// MIT License (see file LICENSE)

// LED is active high
#define GPIO_LED      12
#define GPIO_BUTTONL  25
#define GPIO_BUTTONR  26

static QueueHandle_t queue;

//
```

```
// Button Debouncing task:
//
static void debounce_task(void *argp) {
  unsigned button_gpio = *(unsigned*)argp;
  uint32_t level, state = 0;
  uint32_t mask = 0x7FFFFFFF;
  int event, last = -999;

  for (;;) {
    level = !digitalRead(button_gpio);
    state = (state << 1) | level;
    if ( (state & mask) == mask )
      event = button_gpio;  // Press
    else
      event = -button_gpio; // Release

    if ( event != last ) {
      if ( xQueueSendToBack(queue,&event,1) == pdPASS )
        last = event;
    }
    taskYIELD();
  }
}

//
// Hydraulic Press Task (LED)
//
static void press_task(void *argp) {
  static const uint32_t enable = (1 << GPIO_BUTTONL)
    | (1 << GPIO_BUTTONR);
  BaseType_t s;
  int event;
  uint32_t state = 0;

  // Make sure press is OFF
  digitalWrite(GPIO_LED,LOW);

  for (;;) {
    s = xQueueReceive(
      queue,
      &event,
      portMAX_DELAY
    );
    assert(s == pdPASS);

    if ( event >= 0 ) {
      // Button press
      state |= 1 << event;
    } else {
      // Button release
      state &= ~(1 << -event);
    }

    if ( state == enable ) {
      // Activate press when both
      // Left and Right buttons are
      // pressed.
      digitalWrite(GPIO_LED,HIGH);
    } else {
      // Deactivate press
      digitalWrite(GPIO_LED,LOW);
    }
  }
}

//
// Initialization:
//
void setup() {
  int app_cpu = xPortGetCoreID();
  static int left = GPIO_BUTTONL;
  static int right = GPIO_BUTTONR;
  TaskHandle_t h;
  BaseType_t rc;

  delay(2000);          // Allow USB to connect
  queue = xQueueCreate(40,sizeof(int));
  assert(queue);

  pinMode(GPIO_LED,OUTPUT);
  pinMode(GPIO_BUTTONL,INPUT_PULLUP);
  pinMode(GPIO_BUTTONR,INPUT_PULLUP);

  rc = xTaskCreatePinnedToCore(
    debounce_task,
    "debounceL",
    2048,     // Stack size
    &left,    // Left button gpio
    1,        // Priority
    &h,       // Task handle
    app_cpu   // CPU
  );
```

```
0104:   assert(rc == pdPASS);
0105:   assert(h);
0106:
0107:   rc = xTaskCreatePinnedToCore(
0108:     debounce_task,
0109:     "debounceR",
0110:     2048,     // Stack size
0111:     &right,   // Right button gpio
0112:     1,        // Priority
0113:     &h,       // Task handle
0114:     app_cpu   // CPU
0115:   );
0116:   assert(rc == pdPASS);
0117:   assert(h);
0118:
0119:   rc = xTaskCreatePinnedToCore(
0120:     press_task,
0121:     "led",
0122:     2048,     // Stack size
0123:     nullptr,  // Not used
0124:     1,        // Priority
0125:     &h,       // Task handle
0126:     app_cpu   // CPU
0127:   );
0128:   assert(rc == pdPASS);
0129:   assert(h);
0130: }
0131:
0132: // Not used:
0133: void loop() {
0134:   vTaskDelete(nullptr);
0135: }
```

Listing 3-2. The press.ino demonstration of a hydraulic press controller.

The setup() is much like the last example, except that another button is configured (line 93), and two debounce tasks are created (lines 95 to 117). These tasks are created with an argument provided (lines 99 and 111), to indicate which GPIO button to sample. The debounce_task() code picks up the GPIO number in line 15.

The press_task() in lines 39 to 75, manages the hydraulic press (the LED displays the state of the press). The press is deactivated at startup (line 47), and the forever loop in lines 49 to 74 carry out the event processing. The data type for the event is the integer (int) type, and is received in lines 50 to 54. This receive call blocks forever until an event is received due to the use of portMAX_DELAY.

Event management is somewhat different this time because the code must determine which button was activated or released. Variable state (line 44) is used to hold a 1-bit if the corresponding button is pressed. When a button press event is received, the bit (by GPIO number) is turned on (line 59). A negative event number indicates a "not pressed" event, and the corresponding GPIO bit is disabled in line 62. The constant named enable (lines 40 and 41) defines the value constant that represents both buttons pressed. When this state is achieved (line 65) the press is activated in line 69. With any hint of a button release, a button release will be received and immediately deactivate the press in line 72.

If you lack two push buttons, you can use a breadboard and simulate a button press using a Dupont wires connecting or disconnecting. When you flash and run the program, the initial state of the LED in this demonstration is off. To light the LED (activate the press), both GPIOs using a button (or wires), must be pressed (wires grounded).

Safety Improvement
The code previously presented is potentially unsafe for operating a dangerous press. Can you spot the problem? The debounce_task() queues the event using a wait time of 1 tick (line 29). If this was a button release event, then it might fail to queue if the queue was full. A button release event is safety critical for deactivating the press. A fail after 1 ms or the delay alone may be unacceptable. This demonstration uses a queue length of 2, so some sort of guarantee would be an improvement.

Listing 3-3 illustrates press2.ino, modified for additional safety. The modified task takes corrective action if it finds the queue full and the event is a button release (lines 32 through 40). This provides us with an opportunity to apply the xQueueReset() API function.

```
0001: // press2.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // LED is active high
0005: #define GPIO_LED      12
0006: #define GPIO_BUTTONL  25
0007: #define GPIO_BUTTONR  26
0008:
0009: static QueueHandle_t queue;
0010: static const int reset_press = -998;
0011:
0012: //
0013: // Button Debouncing task:
0014: //
0015: static void debounce_task(void *argp) {
0016:   unsigned button_gpio = *(unsigned*)argp;
0017:   uint32_t level, state = 0;
0018:   uint32_t mask = 0x7FFFFFFF;
0019:   int event, last = -999;
0020:
  for (;;) {
    level = !digitalRead(button_gpio);
    state = (state << 1) | level;
    if ( (state & mask) == mask )
      event = button_gpio;  // Press
    else
      event = -button_gpio; // Release

    if ( event != last ) {
      if ( xQueueSendToBack(queue,&event,0) == pdPASS ) {
        last = event;
      } else if ( event < 0 ) {
        // Queue full, and we need to send a
       // button release event. Send a reset_press
       // event.
        do {
          xQueueReset(queue); // Empty queue
        } while ( xQueueSendToBack(queue,&reset_press,0) != pdPASS );
        last = event;
      }
    }
    taskYIELD();
  }
}

//
// Hydraulic Press Task (LED)
//
static void press_task(void *argp) {
  static const uint32_t enable = (1 << GPIO_BUTTONL)
    | (1 << GPIO_BUTTONR);
  BaseType_t s;
  int event;
  uint32_t state = 0;

  // Make sure press is OFF
  digitalWrite(GPIO_LED,LOW);

  for (;;) {
    s = xQueueReceive(
      queue,
      &event,
      portMAX_DELAY
    );
    assert(s == pdPASS);

    if ( event == reset_press ) {
      digitalWrite(GPIO_LED,LOW);
      state = 0;  printf("RESET!!\n");
      continue;
    }

    if ( event >= 0 ) {
      // Button press
      state |= 1 << event;
    } else {
      // Button release
      state &= ~(1 << -event);
    }

    if ( state == enable ) {
      // Activate press when both
      // Left and Right buttons are
      // pressed.
      digitalWrite(GPIO_LED,HIGH);
    } else {
      // Deactivate press
      digitalWrite(GPIO_LED,LOW);
    }
  }
}

//
// Initialization:
//
void setup() {
  int app_cpu = xPortGetCoreID();
  static int left = GPIO_BUTTONL;
  static int right = GPIO_BUTTONR;
  TaskHandle_t h;
  BaseType_t rc;

  delay(2000);          // Allow USB to connect
  queue = xQueueCreate(2,sizeof(int));
  assert(queue);

  pinMode(GPIO_LED,OUTPUT);
  pinMode(GPIO_BUTTONL,INPUT_PULLUP);
  pinMode(GPIO_BUTTONR,INPUT_PULLUP);

  rc = xTaskCreatePinnedToCore(
    debounce_task,
0113:     "debounceL",
0114:     2048,     // Stack size
0115:     &left,    // Left button gpio
0116:     1,        // Priority
0117:     &h,       // Task handle
0118:     app_cpu   // CPU
0119:   );
0120:   assert(rc == pdPASS);
0121:   assert(h);
0122:
0123:   rc = xTaskCreatePinnedToCore(
0124:     debounce_task,
0125:     "debounceR",
0126:     2048,     // Stack size
0127:     &right,   // Right button gpio
0128:     1,        // Priority
0129:     &h,       // Task handle
0130:     app_cpu   // CPU
0131:   );
0132:   assert(rc == pdPASS);
0133:   assert(h);
0134:
0135:   rc = xTaskCreatePinnedToCore(
0136:     press_task,
0137:     "led",
0138:     2048,     // Stack size
0139:     nullptr,  // Not used
0140:     1,        // Priority
0141:     &h,       // Task handle
0142:     app_cpu   // CPU
0143:   );
0144:   assert(rc == pdPASS);
0145:   assert(h);
0146: }
0147:
0148: // Not used:
0149: void loop() {
0150:   vTaskDelete(nullptr);
0151: }
```

Listing 3-3. Listing of press2.ino with added safety measure:

When a button release can't be queued, a loop in lines 36 to 38 is launched. This clears the queue by calling xQueueReset(). Once cleared, a special reset_press (declared in line 10) event is queued. But here again, timing can be an issue and the queue might be full again (line 38). The time to wait parameter is provided as zero so that if the queue is found full again, it will fail immediately. This causes the loop to continue until a press_release event is successfully queued.

The button press event is also queued in line 30 with a wait time of zero in this program. This permits an immediate fail if the event can't be queued. A button press event can be retried with the next iteration of the loop (notice that the variable last is not updated unless the event was indeed queued in line 31). But a button release is safety-critical.

Line 104 created the queue with a depth of 2. Even when set to a depth of 1, I found it difficult to trigger a reset condition during testing. To make it easier to cause a queue full condition, set the depth to 1 and add a delay(10) ahead of line 60 (this reduces the frequency of queue receive calls). Then place a printf("RESET\n") in after line 39 so that you can prove that the code was provoked, by looking at the Serial monitor output. Then by going ape on the buttons (or scratching the Dupont wires), you should be able to cause a few press_reset events to occur, to prove that the code is working.

The Temptation to Optimize
Oh, how tempting programmers find it to optimize the code! Here it was tempting to just queue the button release event instead of creating a special press_reset event. But since we are clearing the event queue, event data is lost. The press_task() maintains a state variable (line 54). So if by clearing the queue causes a left button release event to be lost, after the state variable logged that the left was pressed, then the state would still remember that left is in the pressed state (even with the queue cleared). Then if the right button later registered a press event by itself, the hydraulic press would activate, even though the operator was no longer pushing the left button! Sending a reset_press event instead permits the press_task() *to reset the state* as well as to deactivate the press (lines 67 to 71).

Informational API
Some other FreeRTOS functions provide information about a queue. These have limited value depending upon how they are applied. For example, uxQueueMessagesWaiting() returns how many items are found within a queue. However, when used with multiple receiving tasks, there is a race condition. By the time you go to receive from the queue, another task may have consumed the queued items and reduced the count to zero.

API function uxQueueSpacesAvailable() is a similar call. When you have multiple tasks queuing to the same queue, a race condition exists when you rely on the returned space count. By the time you know how many spaces are available, the queue could be full by the time you try to add the item.

In general, if you need to use information APIs like these, it might be a sign that your design needs to be improved.

Peeking at the Queue
Sometimes it is convenient to peek at the next item in the queue, before actually receiving it. This usually has to do with the nature of the code processing the entries. Perhaps something special needs to happen before the actual processing of certain items. Care should

be used with this type of API function because it may also suggest poor design. It is valid when the calling task is the only one receiving from the queue. Otherwise, it is subject to race conditions.

The xQueuePeek() function is called with the same arguments as xQueueReceive(). As "peek" implies, it differs by not removing the item that was returned.

Variable Length Items

The advanced user may be dissatisfied with the fixed-length data item requirement. What if the programmer wants to send a variable length line of text through the queue? This limitation *can* be overcome with some added programmer responsibility, by sending data items by pointer. A text line can be sent by allocating memory for the line and copying the text into it (perhaps by using the library function strdup()). The responsibility then shifts to the receiving end – it must know to free() the received item when it is finished with the item.

This becomes problematic if your programming team members sometimes queue up string constants instead of dynamically allocated strings. The code will compile but be disastrous when the received code tries to call free() on the string constant pointer.

Another problem occurs if xQueueReset() is used. All the queued items (pointers) will be lost during the reset, so no free() will be performed. This is a memory leak and could lead to exhausting the heap.

Further, C++ objects like std::string *cannot* be sent as data items. The FreeRTOS API is written in the C language and knows nothing about copy constructors or destructors. You could, however, create your own C++ class to work around these issues with care.

Whenever queuing items by pointer, the caller must also be conscious of the *lifetime of the object*. If you queue a pointer from the current stack frame, then this is a recipe for disaster. The queued item may cease to exist by the time the pointer is received because the sending function has already returned from its stack frame, or the stack object went out of scope. The advanced programmer knows these things but students need to be aware to avoid early subsequent hair loss.

Interrupt Processing

It is too early to cover FreeRTOS interrupt handling. But at this early juncture, it is useful to warn the enthusiast that there are special considerations for queuing data items from within an ISR. It is also important to know that it can be done. Chapter 9 Interrupt Processing will cover this. For now, simply know that you must *never* call API functions like xQueueSendToBack() from inside of an ISR. FreeRTOS makes special provision for interrupt processing using special APIs, with the suffix "FromISR" appended to the name. From within an ISR, you could call xQueueSendToBackFromISR() for example. However, this function also requires a special argument, which requires explanation.

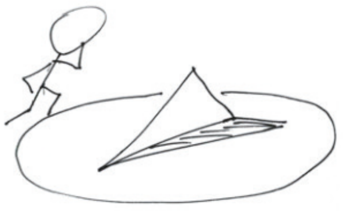
Summary
This chapter has introduced the reader to one of the most important communication concepts within FreeRTOS. Queues provide a clear and able buffer between tasks to communicate data items safely and atomically. Queued data items are copied from the item storage to the queue storage so that the queue mechanism is not subject to lifetime considerations for variables on the stack. This provides added safety for non-pointer types.

Queues also respect the FreeRTOS task priority design. When two tasks are waiting to receive an item from the same queue, FreeRTOS will guarantee that the highest priority task receives the item first. Priority is baked into FreeRTOS.

Next, we look at the FreeRTOS timer related API. You have already used the Arduino delay() function, but tasks often have more particular time management needs.

Exercises
1. Can any FreeRTOS function be called from within an ISR, and if not, why?
2. Which FreeRTOS function can be used to examine the next item in the queue without removing it?
3. When calling a FreeRTOS function that accepts a timeout argument, what value specifies that the caller does not want a timeout.
4. Where is the size of the queue's data item specified?
5. What size requirement does the received data item have when receiving from a queue.
6. How do you receive a data item from a queue without having its execution blocked?
7. What return value does xQueueReceive() return if the call timed out?
8. Which end does the xQueueReceive() always return? Front or Back?

Chapter 4 • Timers



Is it time?

So much in the physical world around us depends upon timing. Waiting for a date in the coffee shop is one example. If the date fails to arrive, then the patron gives up, pays the bill and leaves. Baking with an oven requires that the food cook for the correct amount of time. Pills for medical conditions must be taken every so many hours for the best effectiveness. There are endless examples in life where time plays a role.

This chapter examines the FreeRTOS timer facilities. It's API is straight forward enough but its mastery has traps for the unwary. So buckle up your seat belt as we dive in.

Timer Categories

Within the Arduino framework, you will encounter different types of timer support. These can be generally categorized as:

- Arduino API (examples delay(), millis(), etc.)
- ESP hardware timer API
- FreeRTOS software timer support

You're probably already familiar with the delay() function provided by Arduino platforms. This is often a wrapper function or macro around the function that does the actual work. For example, the ESP32 delay() function is implemented as the FreeRTOS function vTask-Delay():

```
void delay(uint32_t ms)
{
    vTaskDelay(ms / portTICK_PERIOD_MS);
}
```

Given that the normal tick period within the ESP32 is 1 millisecond, this function is equivalent to:

```
void delay(uint32_t ms)
{
    vTaskDelay(ms / 1);
}
```

The ESP32 also provides access to high-resolution hardware timers, which are useful for measuring high-frequency events. That API is covered in the Espressif online documentation[1] or some of the Arduino provided functions like micros().

The remaining category, which is the main subject of this chapter, is the FreeRTOS software timer API.

Software Timers

Within the ESP32 Arduino environment, the macro value configUSE_TIMERS is configured as true, which indicates support for software timers. FreeRTOS provides the following general operations:

1. Create a timer (and receive a handle to it).
2. Re/configure a timer
3. Start/Restart/Stop a timer
4. Delete a timer

Before we examine the mechanics of these operations, let's first review how the timer event is delivered to your task.

The Timer Callback

The problem with timers is that they produce an asynchronous event concerning the task that created it. So while your task executes, or remains blocked by some other resource, the timer event must be delivered when it is time to be triggered. The FreeRTOS solution is to use the user provided callback function of the form:

```
        void my_timer_cb(TimerHandle_t xTimer);
```

When the timer is triggered, your function my_timer_cb is called with the single argument, which is the handle for the timer. Within your function, your code will perform whatever needs to be done, possibly alerting the rest of your application. FreeRTOS's responsibility ends with the calling of the timer callback.

Timer Limitations

While your callback function has full access to your application code and data, there are restrictions on what it is allowed to perform. These restrictions include:

- The execution time should be short.
- The callback code must not enter the *blocked* state.
- No calls to vTaskDelay() (or delay()) are permitted -- these immediately place the calling task into the *blocked* state.
- The callback must observe stack size limitations (configTIMER_TASK_STACK_DEPTH indicates that the affected stack's size is 2048 bytes).

These limitations can be quite problematic. These stem from the fact that the callback itself is made from the "Tmr Svc" task (task number 8 in Chapter 2, Table 2-1). The callback is *not* issued from the task that created it. The "Tmr Svc" task is also known as the "RTOS daemon task", or just "daemon task". Older literature will also refer to it as the "timer service task" before the scope of the task was expanded.

Table 2-1 showed that it had 1468 bytes remaining unused when the task info was captured. So for the ESP32 Arduino, this gives you a ballpark stack availability for your callback. Students should bear in mind that functions like printf() or snprintf() can require considerable stack space and should probably be avoided in the callback.

The daemon task performs many of its functions through the use of a timer command queue (the task and queue are automatically created when the FreeRTOS scheduler is started). So it processes queued commands in addition to performing timing functions. If your callback were to take too long to execute or cause the daemon task to become *blocked*, then other timer services would fail. This type of abuse ruins the unicorn harmony that is maintained within FreeRTOS.

Summarizing then, know that your software timer callback is called from someone else's home (task). Your code is a guest there and must observe "house rules". Also implied is the fact that your callback is asynchronous concerning the task that created the timer. Consequently, some safe form of task-to-task communication is usually needed from the callback.

Timer ID Value

The FreeRTOS software timer API uses an optional entity known as the "timer ID". It is referenced in API function calls as a void pointer type with the name pvTimerID. Supply nullptr when it is not needed. The choice of naming for this facility is unfortunate and confusing.
The user always obtains handles to resources such as tasks, queues, and timers. These handles can be used to manipulate, identify, and delete these resources. Each also has an assigned user-friendly string name. So what is the purpose of this timer ID? The callback already receives the handle of the timer as an argument in the call. Can't you just identify the timer by its handle?

You can indeed identify the timer by handle. But the problem remaining is that the callback function code may be shared among several timer instances because the required action is the same. But each timer is managing a different application resource, like for example a UDP socket. Perhaps the callback is expected to initiate a retransmit of a packet if no response is received within the timeout period. What the callback needs is a pointer to some *associated "user data"* structure. Without this association, the callback would need to use a lookup table.

Rather than call the entity a "timer ID", I prefer to use the more conventional term *"user data"* parameter. To make this concrete, let's use an example. The developer creates his application-specific data structure like s_udp_socket below:

```
struct s_udp_socket {
  int sock;     // UDP socket number
  int retries;  // Count of retries
  ...
};
```

Then, within the timer callback, the data associated with the timer is fetched using the FreeRTOS function pvTimerGetTimerID(), which requires the passed timer handle. The value returned is a void pointer, so you must *carefully* cast it to the correct (struct) s_udp_socket pointer type. This gives the callback immediate access to the socket that the timeout occurred on.

```
void udp_timeout_cb(TimerHandle_t handle) {
  s_udp_socket *user_data = (s_udp_socket*) pvTimerGetTimerID(handle);

  ++user_data->retries;
  issue_retransmit(user_data->sock);
  ...
}
```

Shortly, you will see that this user data association is often established at timer create time. However, using the FreeRTOS function xTimerSetTimerID(), it can be established or changed after the fact. When you do not need this functionality, supplying nullptr is permitted. FreeRTOS only registers the value for you but otherwise does not use it.

Abusing Timer ID
Sometimes, the association required in an application is simply an integer or unsigned value. The developer might consider that using a pointer to a structure is overkill. Using the previous example, if you only needed the socket number, then you could use that directly in place of a pointer. I'll refer to this practice as "abusing the pointer".

First, consider the void pointer's size. For the ESP32, a void pointer is a 32-bit address. Knowing this, you can get away with using a 32-bit integer or unsigned value with appropriate casting.

The previous example's callback could have been written like this instead:

```
void udp_timeout_cb(TimerHandle_t handle) {
  int sock = (int)pvTimerGetTimerID(handle);

  issue_retransmit(sock);
  ...
}
```

This type of short-cut is poor economy in my opinion. Applications normally tend to get more complex as new requirements are identified. Seldom do they get simpler. If you use the structure approach, it is a simple matter to add a new structure member when necessary. If you abused the void pointer and now need to use the structure approach, you will have to hunt down all locations where this abuse was applied and revise it. In large applications, this can be a nightmare. I would only use the practice as a last resort when every last byte of storage must be grasped.

Timer Types
There are two basic software timer types within FreeRTOS:

- The one-shot timer.
- The auto-reload timer.

The one-shot timer as the name suggests is expected to trigger once. The auto-reload timer is a timer that triggers automatically at specific intervals and re-arms itself. It will continue to run on its own until it has been stopped.

Timer States
The FreeRTOS timer exists in two different states:

- **Dormant:** the timer has not been activated/reactivated.
- **Running:** the timer is active.

When the timer is created, it is initialized in the dormant state. It is not made active until the timer has been started/restarted.

The state diagram for one-shot timers is shown in Figure 4-1. The create function xTimerCreate() immediately puts the timer into the dormant state. When one of the xTimerStart() functions is called, the timer then enters the running state until the timer expires. Upon expiry, the callback is performed and the timer returns to the dormant state.

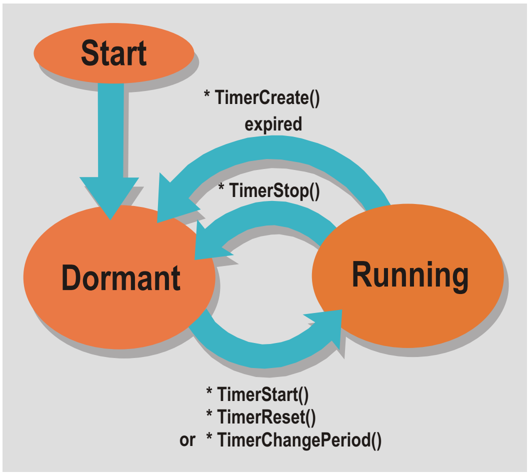


Figure 4-1. The states of a one-shot timer.

The states of the auto-reload timer are illustrated in Figure 4-2. After the timer is created, it is put into the dormant state, until one of the start functions are called. Once started, the timer enters the running state, and reenters the running state when it expires, after performing the callback. Only stopping the timer returns it to the dormant state.

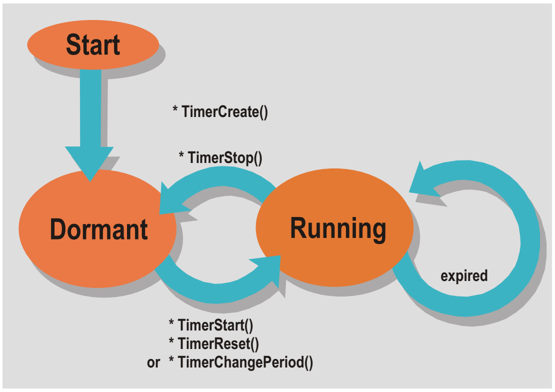


Figure 4-2. The states of the auto-reload timer.

Create Static Timer

Let's now see how to create a timer. When supported, to create a timer statically, you must allocate the storage for a type StaticTimer_t (at the present release it is 40 bytes in size). This object will manage the timer, once configured. Save the returned handle:

```
StaticTimer_t tmr1_obj; // Timer object

struct s_user_data {
  members...
} socket1;
TimerHandle_t h;

h = xTimerCreateStatic( // handle returned
  "my_timer1",          // friendly name
  timer_period,         // Period in ticks
  pdFALSE,              // No reload
  &socket1,             // User data pointer
  my_timeout_cb,        // Timer callback
  &tmr1_obj             // Static timer object
);
assert(h);              // Check h != nullptr
```

The string name is simply a friendly string name of the resource that you are creating. It is not otherwise used by FreeRTOS. The timer_period is specified in ticks. If you'd prefer to specify the time in milliseconds, then use the pdMS_TO_TICKS() macro instead. Be careful about the precision involved in the macro call (specifying a time less than 1 millisecond, will result in supplying 0 ticks for the ESP32).

The argument pdFALSE specifies that this will be a one-shot timer. Supplying pdTRUE creates an auto-reload timer.

The argument &socket1, is a pointer to the user data. If not required, supply nullptr. Argument my_timeout_cb is the name of your callback function. This is the function that will be called from the daemon task when the timer expires. The last parameter is the pointer to the statically allocated timer object.

Remember that a timer is always created in the dormant state and requires another step to activate it.

Create Dynamic Timer
The ESP32 Arduino environment does support the creation of dynamically allocated timers. The calling requirements are the same as before, except that FreeRTOS creates the timer object for you:

```
struct s_user_data {
  members...
} socket1;
TimerHandle_t h;

h = xTimerCreate( // handle returned
  "my_timer1",    // friendly name
  timer_period,   // Period in ticks
  pdFALSE,        // No reload
  &socket1,       // User data pointer
  my_timeout_cb,  // Timer callback
);
assert(h);
```

Like the static timer, the timer is always created in the dormant state and requires another step to activate it.

Activating the Timer
There are multiple ways to activate the timer:

1. xTimerStart()
2. xTimerReset()
3. xTimerChangePeriod()

The first two API functions are in fact equivalent. From a documentation perspective, xTimerStart() is preferred when the timer is known to be dormant. But the function will perform a reset if it is found to be active. The xTimerReset() function is preferred when the timer is known to be running. But the call will otherwise start a dormant timer. These differ only by name.

```
        BaseType_t xTimerStart(
          TimerHandle_t xTimer,
          TickType_t xTicksToWait
        );

        BaseType_t xTimerReset(
          TimerHandle_t xTimer,
          TickType_t xTicksToWait
        );
```

Each of these requires the handle to the created timer. The xTicksToWait parameter might be puzzling when first encountered. These API functions create and send a message to the daemon task through a message queue. If the queue happens to be full, the API call needs to know how long to wait. Zero causes an immediate fail if the queue is full, and a specified time will block for a time, while the value portMAX_DELAY will not return until the request has been queued. The possible return values are pdPASS or pdFAIL.

Within the ESP32 Arduino environment, the macro configTIMER_QUEUE_LENGTH is defined as 10. This configures the maximum depth of the timer command queue.

The last function of interest is xTimerChangePeriod() and can be used on a dormant or running timer. It also requires the timer's handle, the new timer period to use, and the xTicksToWait parameter like the previous functions. This API call queues the request through a message queue.

```
        BaseType_t xTimerChangePeriod(
          TimerHandle_t xTimer,
          TickType_t xNewPeriod,
          TickType_t xTicksToWait
        );
```

Demonstration

The demonstration for this chapter uses a class named AlertLED, that drives an LED in an eye-catching manner to indicate some critical error or fault. The class instance is created with the LED GPIO number to be driven and the blink period (1000 ms by default). When activated to display an alert, the LED flashes quickly 5 times, then pauses for the remainder of the period before repeating.

Figure 4-3 illustrates the scope trace of the LED drive signal when active. The LED is driven high (active high) five times, and then the LED is off for the remainder of the period. Figure 4-4 illustrates the wiring used for this demonstration.

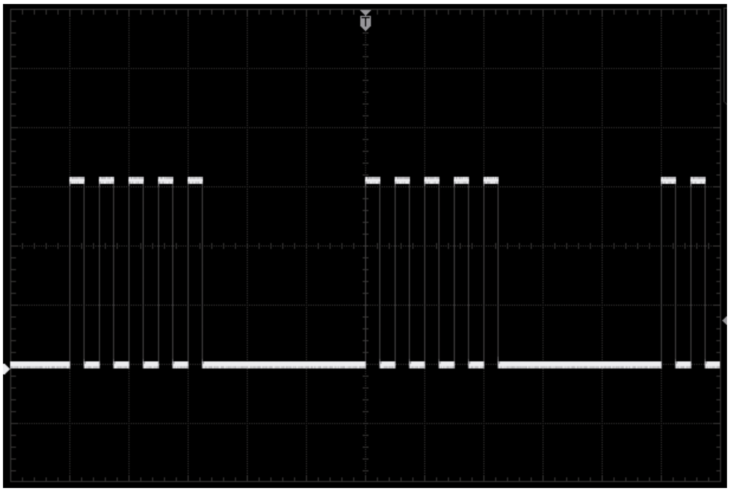


Figure 4-3. GPIO drive for alert LED, horizontal is 200 ms / div.

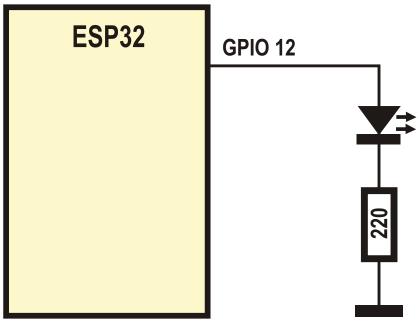


Figure 4-4. The wiring used for the alertled.ino program.

By now, some might be muttering that this could have been easily done with a task. In other words, what is the payoff for using this timer API? Indeed, it could be written as a task, so where is the advantage? How would the two methods compare if you needed to support 32 LED indicators?

For the task-based approach you would need 32 separate task stacks. For the timer-based approach, the timer callback makes use of the one daemon task's stack. The stack provided is re-used for each timer making a callback. The timer approach is memory frugal!

The full program listing is provided in Listing 4-1. But let's examine it in small chunks. The class named AlertLED is presented first:

```
class AlertLED {
  TimerHandle_t     thandle = nullptr;
  volatile bool     state;
  volatile unsigned count;
  unsigned          period_ms;
0015:   int             gpio;
0016:
0017:   void reset(bool s);
0018:
0019: public:
0020:   AlertLED(int gpio,unsigned period_ms=1000);
0021:   void alert();
0022:   void cancel();
0023:
0024:   static void callback(TimerHandle_t th);
0025: };
```

The constructor (line 20) accepts a GPIO number for the LED, and a time period in milliseconds (which defaults to 1000 ms). The method AlertLED::alert() (line 21) is called when you want to activate the alert mode, or AlertLED::cancel() (line 22) when you want to disable it. The AlertLED::reset() (line 17) is internal to the class and is used to reset the state of the object members.

The method AlertLED::callback() (line 24) is registered to the class as a static method. This means that there is no attached object when it is called. It is much like calling a regular C function except for the funky C++ name. We'll look at the callback in greater detail later.

AlertLED Constructor

The class constructor is nothing fancy except that it initializes the data members to reflect the configured GPIO (line 33), the configured time period (line 34), configures the GPIO as an output (line 35) and initializes the initial GPIO output level (line 36). The timer handle has already been initialized in the listing line 11 (as nullptr). At this point, no FreeRTOS timer exists yet.

```
0032: AlertLED::AlertLED(int gpio,unsigned period_ms) {
0033:   this->gpio = gpio;
0034:   this->period_ms = period_ms;
0035:   pinMode(this->gpio,OUTPUT);
0036:   digitalWrite(this->gpio,LOW);
0037: }
```

AlertLED Instance

In Listing 4-1, the instance of the class is created on line 100.

```
0100: static AlertLED alert1(GPIO_LED,1000);
```

By the time that the setup() and loop() functions execute, the class object alert1 has already been initialized (constructed) and is standing by.

In the setup() function (line 106), the alert1.alert() method is called to start the alert display. Let's now examine what the method call performs.

AlertLED::alert() Method
When AlertLED::alert() is called, it first checks to see if it has a timer handle (line 53). If it doesn't, then a timer is created in lines 54 to 59, with the handle checked in line 60. Then internal method AlertLED::reset() is called (line 62) to re/initialize the object to a known state. If the timer is dormant, it now needs startup by calling xTimerStart() in line 63.

```
0051: void AlertLED::alert() {
0052:
0053:   if ( !thandle ) {
0054:     thandle = xTimerCreate(
0055:       "alert_tmr",
0056:       pdMS_TO_TICKS(period_ms/20),
0057:       pdTRUE,
0058:       this,
0059:       AlertLED::callback);
0060:     assert(thandle);
0061:   }
0062:   reset(true);
0063:   xTimerStart(thandle,portMAX_DELAY);
0064: }
```

Calling xTimerStart() puts the timer into a running state. The argument pdMS_TO_TICKS(period_ms/20) (line 56) indicates that the next timer event is 1000/20 ms = 50 ms from the start time (line 63). When that time arrives, the AlertLED::callback() method will be invoked.

The AlertLED::callback
The callback was established as AlertLED::callback() in line 59, of the AlertLED::alert() method. That timer was configured as an auto-reload timer (line 57). So when the timer is triggered, the callback is made.

The first step within the callback is to locate the class object using the Timer ID (user data). Line 80 gets the user data pointer and casts it to the class object pointer. This pointer was established in line 58 of the AlertLED::alert() method call. For the benefit of C language programmers, the "this" keyword used in line 58 points to the object instance. To check the association, the assert() macro is applied in line 82).

Once the AlertLED class pointer obj is known, the next event for the object is determined. First, the state of the LED is toggled in data member state (line 83). Then this is written out to the GPIO pin (line 84).

The count of the number of callbacks is maintained in the data member obj->count, which is incremented (line 86). After incrementing the counter, control flows out of the module initially, until the count reaches 9 in line 89. Lines 90 to 92 then change the timer period to half a period plus a small adjustment. The adjustment accounts for the off period of the on/off cycle of the LED pulse. This corrects for the full period length. During this time, the

LED is off until that time expires.

At that point, the count will match 10 in line 86. The code that follows then resets the class member state and counter (line 87) and changes the timer period back to obj->period_ms/20 to begin a new cycle.

```
0079: void AlertLED::callback(TimerHandle_t th) {
0080:   AlertLED *obj = (AlertLED*)pvTimerGetTimerID(th);
0081:
0082:   assert(obj->thandle == th);
0083:   obj->state ^= true;
0084:   digitalWrite(obj->gpio,obj->state?HIGH:LOW);
0085:
0086:   if ( ++obj->count >= 5 * 2 ) {
0087:     obj->reset(true);
0088:     xTimerChangePeriod(th,pdMS_TO_TICKS(obj->period_ms/20),portMAX_
DELAY);
0089:   } else if ( obj->count == 5 * 2 - 1 ) {
0090:     xTimerChangePeriod(th,
0091:       pdMS_TO_TICKS(obj->period_ms/20+obj->period_ms/2),
0092:       portMAX_DELAY);
0093:     assert(!obj->state);
0094:   }
0095: }
```

Stopping the Alert
To stop the AlertLED class from driving the LED, the method AlertLED::cancel() is called. All it has to do is to stop the timer (line 71) and write a low to the LED (line 72 of Listing 4-1).

setup() and loop()
The setup() routine starts the alert in line 113 by calling alert1.alert(). Uncomment the delay(2000) if you plan to add any debug print() calls so that the USB controller has time to connect to the TTL to serial chip.

To prove that the alert start and stop works as expected, the loop() function has code to start and stop the alert. The counter loop_count (line 101) tracks how many times the loop() function has been called. Once the count exceeds 50 (line 119), the alert is stopped by invoking alert1.cancel() in line 120. But loop() will continue to be called, and eventually, the count will reach 70 in line 112. When that occurs, the alert is re-enabled in line 113, where the count is also reset.

```
0001: // alertled.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // LED is active high
0005: #define GPIO_LED      12
```

```

//
// AlertLED class to drive LED
//
class AlertLED {
  TimerHandle_t thandle = nullptr;
  volatile bool     state;
  volatile unsigned count;
  unsigned          period_ms;
  int               gpio;

  void reset(bool s);

public:
  AlertLED(int gpio,unsigned period_ms=1000);
  void alert();
  void cancel();

  static void callback(TimerHandle_t th);
};

//
// Constructor:
//  gpio        GPIO pin to drive LED on
//  period_ms   Overall period in ms
//
AlertLED::AlertLED(int gpio,unsigned period_ms) {
  this->gpio = gpio;
  this->period_ms = period_ms;
  pinMode(this->gpio,OUTPUT);
  digitalWrite(this->gpio,LOW);
}

//
// Internal method to reset values
//
void AlertLED::reset(bool s) {
  state = s;
  count = 0;
  digitalWrite(this->gpio,s?HIGH:LOW);
}

//
// Method to start the alert:
//
void AlertLED::alert() {

  if ( !thandle ) {
    thandle = xTimerCreate(
      "alert_tmr",
      pdMS_TO_TICKS(period_ms/20),
      pdTRUE,
      this,
      AlertLED::callback);
    assert(thandle);
  }
  reset(true);
  xTimerStart(thandle,portMAX_DELAY);
}

//
// Method to stop an alert:
//
void AlertLED::cancel() {
  if ( thandle ) {
    xTimerStop(thandle,portMAX_DELAY);
    digitalWrite(gpio,LOW);
  }
}

// static method, acting as the
// timer callback:
//
void AlertLED::callback(TimerHandle_t th) {
  AlertLED *obj = (AlertLED*)pvTimerGetTimerID(th);

  assert(obj->thandle == th);
  obj->state ^= true;
  digitalWrite(obj->gpio,obj->state?HIGH:LOW);

  if ( ++obj->count >= 5 * 2 ) {
    obj->reset(true);
    xTimerChangePeriod(th,pdMS_TO_TICKS(obj->period_ms/20),portMAX_DELAY);
  } else if ( obj->count == 5 * 2 - 1 ) {
    xTimerChangePeriod(th,
      pdMS_TO_TICKS(obj->period_ms/20+obj->period_ms/2),
      portMAX_DELAY);
    assert(!obj->state);
  }
}

```

```
//
// Global objects
//
static AlertLED alert1(GPIO_LED,1000);
static unsigned loop_count = 0;

//
// Initialization:
//
void setup() {
  // delay(2000); // Allow USB to connect
  alert1.alert();
}

void loop() {
  if ( loop_count >= 70 ) {
    alert1.alert();
    loop_count = 0;
  }

  delay(100);

  if ( ++loop_count >= 50 )
    alert1.cancel();
}
```

Listing 4-1. The alertled.ino program listing.

Demo Notes

The callback AlertLED::callback() is invoked from the daemon task using that task's stack. Debugging these callbacks can be a challenge because you only have about 1500 bytes of stack available. That might make debugging with printf() a problem. If you need to debug, you might need to resort to techniques like toggling a separate GPIO and observing it with a scope.

Priority 1

Because the damon task runs at priority 1 (in the ESP32), it is important that priority 1 tasks get some execution time or the timers won't trigger or callbacks execute. If you experience this, check that some other higher priority task is not monopolizing the CPU.

The Class Advantage

To add more alert LEDs to your application, you simply instantiate more instances of the AlertLED class, supplied with appropriate GPIO and period arguments. Each instance will add one AlertLED class in storage consumption and a dynamically allocated timer (with minimal overhead). This is a much lighter memory footprint than creating a task for each LED.

Task Timer API
There are a couple of other time related functions that affects tasks and general use that should receive mention in this chapter:

- vTaskDelay()
- xTaskGetTickCount()
- xTaskDelayUntil()

If you invoke the Arduino function delay(), then you are already making use of vTaskDelay(). Recall that we saw that it is implemented as follows:

```
void delay(uint32_t ms)
{
    vTaskDelay(ms / portTICK_PERIOD_MS);
}
```

The application developer must be careful when specifying delay times. If your application specifies everything in terms of microseconds, for example, expect some porting difficulties. A macro that converts from microseconds to milliseconds may compute zero in some places where zero cannot be tolerated. This boils down to timer resolution. The ESP32 works with millisecond system ticks for the Arduino. On other platforms using FreeRTOS, this may not hold true. For this reason, the FreeRTOS engineers recommend specifying all times in ticks rather than in time units.

xTaskGetTickCount()
Use this is the API function to call to get the FreeRTOS sense of system time.

```
TickType_t xTaskGetTickCount(void);
```

This function returns the number of ticks that have occurred since the FreeRTOS scheduler was started. The scheduler is automatically started for the Arduino environment before setup() is called. Do not confuse this with the function millis().

This tick count will overflow if the FreeRTOS scheduler runs long enough. There is no need to fear problems because FreeRTOS has taken this into account within their kernel code. But if you use this time value for your calculations, you need to account for the possibility of rollover. The time difference in ticks might naively be coded as:

```
TickType_t tick1, tick2, delta;

tick1 = xTaskGetTickCount();
// perform some operations
tick2 = xTaskGetTickCount();

delta = tick2 - tick1;   // Fails if overflow occurs
```

That calculation will fail if the tick timer overflowed between tick1 and tick2. An improved calculation is below:

```
if ( tick1 <= tick2 )
  delta = tick2 - tick1; // Normal time difference
else
  delta = tick2 + (~TickType_t(0) - tick1) + 1;
```

The else expression adds the time lost due to the rollover to the current time tick2. The C++ cast ~TickType_t(0) expression breaks down as:

1. C++ cast the 0 constant as type TickType_t, then
2. the tilda(~) operator inverts all bits making them 1-bits (the maximum value that can be held in the corresponding type).

By subtracting tick1 from the maximum value of the type, we calculate the number of ticks lost due to rollover, minus one. To compensate for that minus one, we add it back at the end. Adding this calculated value to tick2, we arrive at the time difference after the rollover.

Note that this calculation is only valid when there has only been one rollover. Some other measures must be in place if there is the possibility of multiple rollover events.

xTaskDelayUntil()

If you need an event scheduled for precisely a certain time regularly, you don't always need to use a software timer. The function xTaskDelayUntil() might be all you need. This API call permits the caller to defer execution (block a task) until a specific future tick time arrives. To keep the timing precise, argument one is both an input time and an updated output ticks value. Argument two is used to schedule the next wakeup time so many ticks from the argument one value:

```
void vTaskDelayUntil(
  TickType_t *pxPreviousWakeTime,
  TickType_t xTimeIncrement
);
```

Again, the first argument points to a TickType_t argument that acts as both an input to the call and receives an updated time when the function returns. When the call is made, the argument one value is accessed by pointer and added to the time increment value in argument two to compute the future time. This is the value that will be returned through the pointer argument one. Because this is a precise calculation, there can be no time slippage.

Listing 4-2 illustrates the demo program delayuntil.ino, with Figure 4-5 illustrating the wiring. This demonstration uses two tasks:

- task1() -- uses the traditional delay() call.
- task2() -- uses the xTaskDelayUntil() call.

The loop used by task1() is shown in lines 23 to 28:

```
0023:   for (;;) {
0024:     state ^= true;
0025:     digitalWrite(GPIO_LED1,state);
0026:     big_think();
0027:     delay(period);
0028:   }
```

The function big_think() invokes a function that performs a time intensive function (implemented in lines 11 to 15). This emulates some time intensive process in your application. The time used by big_think() will cause the timing of the LED toggle (lines 24 and 25) to slip as time progresses.

Compare that code to the loop used in task2():

```
0037:   TickType_t ticktime = xTaskGetTickCount();
0038:
0039:   for (;;) {
0040:     state ^= true;
0041:     digitalWrite(GPIO_LED2,state);
0042:     big_think();
0043:     vTaskDelayUntil(&ticktime,period);
0044:   }
```

The initial tick time is captured in line 37. After that, every call to vTaskDelayUntil() will ensure that it will block until the next calculated time occurs.

Note. If the calculated event time is less than or equal to the current time, the function xTaskDelayUntil() will return immediately.

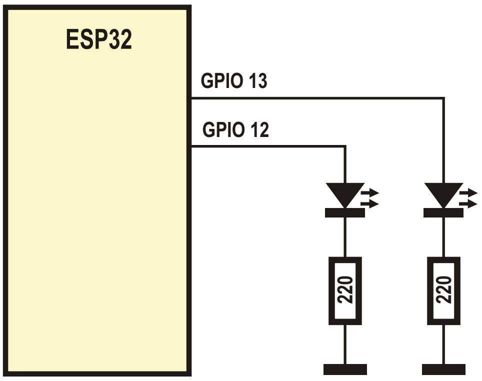


Figure 4-5. Wiring used for program delayuntil.ino.

```
// delayuntil.ino
// MIT License (see file LICENSE)

// LED is active high
#define GPIO_LED1     12
#define GPIO_LED2     13

static volatile bool startf = false;
static TickType_t period = 250;

static void big_think() {

  for ( int x=0; x<40000; ++x )
    __asm__ __volatile__ ("nop");
}

static void task1(void *argp) {
  bool state = true;

  while ( !startf )
    ;

  for (;;) {
    state ^= true;
    digitalWrite(GPIO_LED1,state);
    big_think();
    delay(period);
  }
}

static void task2(void *argp) {
  bool state = true;

  while ( !startf )
    ;

  TickType_t ticktime = xTaskGetTickCount();

  for (;;) {
    state ^= true;
    digitalWrite(GPIO_LED2,state);
    big_think();
    vTaskDelayUntil(&ticktime,period);
  }
}

```

```
0047: //
0048: // Initialization:
0049: //
0050: void setup() {
0051:   int app_cpu = xPortGetCoreID();
0052:   BaseType_t rc;
0053:
0054:   // delay(2000); // Allow USB to connect
0055:   pinMode(GPIO_LED1,OUTPUT);
0056:   pinMode(GPIO_LED2,OUTPUT);
0057:   digitalWrite(GPIO_LED1,HIGH);
0058:   digitalWrite(GPIO_LED2,HIGH);
0059:
0060:   rc = xTaskCreatePinnedToCore(
0061:     task1,
0062:     "task1",
0063:     2048,
0064:     nullptr,
0065:     1,
0066:     nullptr,
0067:     app_cpu);
0068:   assert(rc == pdPASS);
0069:
0070:   rc = xTaskCreatePinnedToCore(
0071:     task2,
0072:     "task2",
0073:     2048,
0074:     nullptr,
0075:     1,
0076:     nullptr,
0077:     app_cpu);
0078:   assert(rc == pdPASS);
0079:
0080:   startf = true;
0081: }
0082:
0083: void loop() {
0084:   delay(50);
0085: }
```

Listing 4-2. Program delayuntil.ino.

Demonstration Observation

When the demonstration begins, the two LEDs will appear to blink in unison. As time wears on, LED1 (task1) will start to slip behind the schedule that LED2 keeps (task2). This occurs because the call to delay() in line 27, is relative to the time it was called. The computation performed in big_think() will cause the call into delay() to be a little bit later with each passing loop.

Task 2 however, uses the vTaskDelayUntil() in a manner that doesn't depend upon the time that the function is called. Being a bit tardy doesn't upset the schedule, provided that the big_think() time is less than the tick time held in variable period.

Figure 4-6 is a photo of the Lolin ESP32 device being used with two LEDs attached. The LEDs shown are using a combined soldered LED and resistor pair as illustrated in Figure 4-7. Using these pairs make for convenient breadboard work.

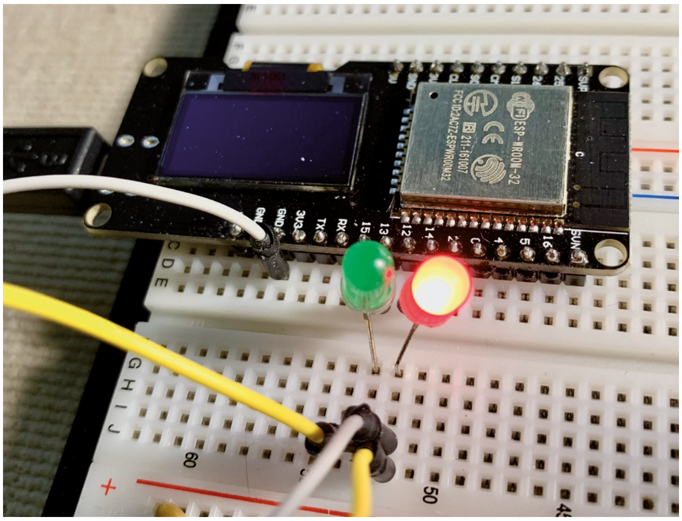


Figure 4-6. The Wemos Lolin ESP32 with LED combos attached for program delayuntil.ino.

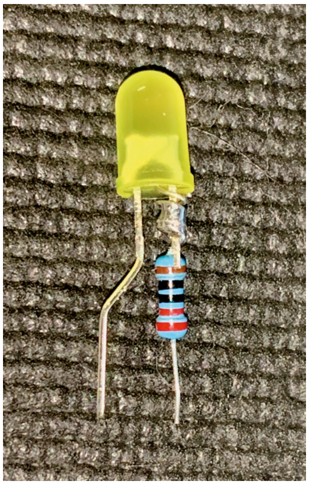


Figure 4-7. The LED and 220-ohm resistor combo is handy for breadboard experiments.

Summary

This chapter introduced the FreeRTOS software timer API and described its advantage over separate tasks to perform the same function. There were some noted disadvantages including the use of a stack of limited size. It also lacks a way to set the execution priority of the callback. Despite these problems, the API remains useful for many purposes. If the needs are simple enough, the xTaskDelayUntil() can sometimes be used instead.

One area that we still need to satisfy is the category of task synchronization. The next chapter will explore semaphores, giving us some badly needed synchronization.

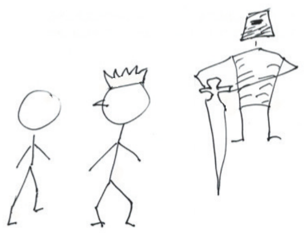
Exercises

1. Can a timer callback issue blocking calls like vTaskDelay()?
2. What state is a timer created in?
3. Which stack does a timer callback make use of?
4. What resource do some of the timer functions make use of?
5. What is the FreeRTOS "Timer ID value" essentially?
6. What is the requirement for safely abusing a pointer to carry an integer value?

Web Resources

[1] "Timer." ESP. Accessed December 27, 2019.
https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/timer.html.

Chapter 5 • Semaphores



Black Knight: "None shall pass!"
King Arthur: "It seems that we are dealing with a binary semaphore!"

A semaphore provides a means of access control or synchronization in multitasking environments. FreeRTOS provides binary semaphores, counting semaphores and mutexes. This chapter explores the first two – binary and counting semaphores, with Arduino demonstration programs.

Semaphore Types

Let's review the overall characteristics of the two semaphore types that FreeRTOS provides: binary and counting semaphores.

Binary Semaphores

As the name "binary" implies, this is a synchronization primitive with two possible states:

- empty (not "given")
- full ("given")

The use of a binary semaphore is fairly simple – one task will block attempting to take a semaphore. The operation blocks the task's execution when the semaphore is found "empty" (not "given"). Another task will "give" the semaphore, which then unblocks the former task. In this manner, the two tasks synchronize their execution.

But questions remain – what state is the binary semaphore created in? In FreeRTOS, a binary semaphore is always created empty (not "given"). This is worth memorizing. This behaviour allows a binary semaphore to be created and used as a "barrier". Your setup() routine might first create a task but not want it to proceed passed some initial point until some other initialization has completed. The setup() task can "give" the semaphore after completion, allowing the created task to proceed.

What happens if no task has "taken" a binary semaphore and another "give" is performed on it? This returns an error (pdFAIL). But if your goal is simply to make sure that there is no blocking (no barrier), the error can safely be ignored in this case.

The binary semaphore is like a FreeRTOS queue with a depth of one item. The distinction of the semaphore is that the queued *value* is unimportant. The queue is either full or empty. When the queue is full, the "give" fails because the queue is full. Likewise, similar to receiving from an empty queue, the "take" operation can error, block, or timeout depending upon the calling parameters.

There is another important characteristic to note – the binary semaphore can only block and notify one task. A TCP/IP program, for example, may want to have several tasks wait until the WiFi has initialized and attached to the access point (AP). A binary semaphore can only be used to block and notify one task. The FreeRTOS event group is a better choice for that example, which is discussed in a later chapter.

Counting Semaphores
There is also the FreeRTOS counting semaphore. The counting semaphore can be used to control access to several resources and is usually used in one of two ways:

- Counting up from zero (all are initially "taken").
- Counting down from some initial count (all are initially "available").

Both uses can be thought of as limit control devices, except that one counts up and the other counts down. Giving the semaphore increases the count while taking from the semaphore reduces it.

In the first case, the count is initialized to zero. This means that no resources can be "taken" because the resource count has been exhausted. Only after a number of gives have taken place, can takes later succeed. A take attempted on an empty counting semaphore will block (or timeout blocking).

When initialized with a maximum value, tasks can "take" from that semaphore until the count reaches zero. When the count reaches zero, a "take" operation will block until another task gives the semaphore (or times out).

FreeRTOS also provides a function named uxSemaphoreGetCount() that will return the current value of the counting semaphore. Care should be used with this function because the count can change between the time the semaphore was taken and the time that the count was fetched (a race condition).

Binary Semaphore Demonstration
To illustrate the synchronizing capability of the binary semaphore, the program in Listing 5-1 named hcsr04.ino is presented.[1] This program doesn't need a binary semaphore because it can be accomplished in one task. However, it was split into two tasks to demonstrate the synchronization utility of the binary semaphore.

The demonstration program uses one task to ping the environment with the HC-SR04 ultrasonic module. The optional LED wired to GPIO 12 provides a visual indication of when the ping occurs. The second task is used to synchronize the ping task exactly once per second ping.

Change the macro USE_SSD1306 in line 5 to zero if you are not using the SSD1306 OLED (as present on the Lolin ESP32). The program output can be viewed without the OLED using the Arduino Serial Monitor. The GPIO numbers can also be customized in lines 9 through 11 if you want. The wiring for this project is shown in Figure 5-1.

The HC-SR04 module has two ultrasonic transducers – one for transmitting the ping and the other for receiving the echo. The PCB module provides signal conditioning opamps and is controlled by a very small microcontroller. The ESP32 requests a ping by setting the Trigger line high for a minimum of 10 µsec (long enough for the tiny microcontroller to notice). Then the ESP32 returns the signal line to low and then listens for a pulse from the Echo line. The ESP32 times the duration of the received pulse and from that calculates the distance.

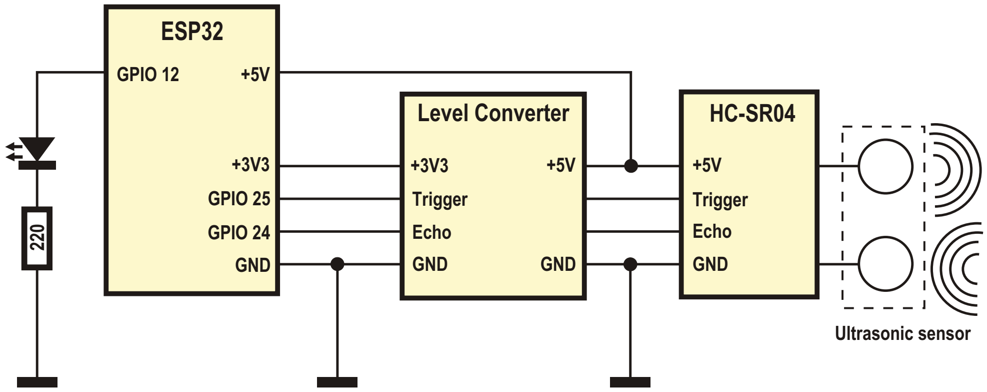


Figure 5-1. Schematic for the hsr05.ino demo program.

One complicating factor in this project is that the HC-SR04 module is a 5-volt device. The ESP32 uses +3.3 volts and does not have 5-volt tolerant GPIOs. Therefore, we must perform signal level translations between the two. In the middle of Figure 5-1 is perhaps the simplest solution to this problem. This small PCB is made out of MOSFETs and resistors to perform the necessary voltage translations. It is available on eBay and elsewhere under names like "4-bit bidirectional voltage-level translator". Module images are shown in Figures 5-2 and 5-3. For this project, we only use two of the four signal lines. Just leave the unused lines unconnected (or tie one end of each line to ground to keep them from floating). An explanation of how this voltage translation works can be seen at the link.[2]

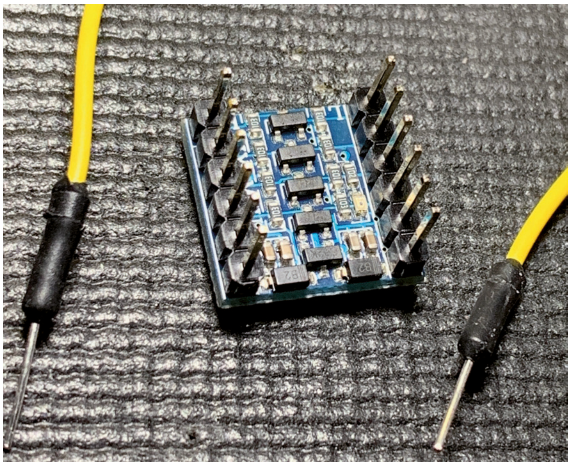


Figure 5-2. The underside of the 3/5 volt level converter PCB.

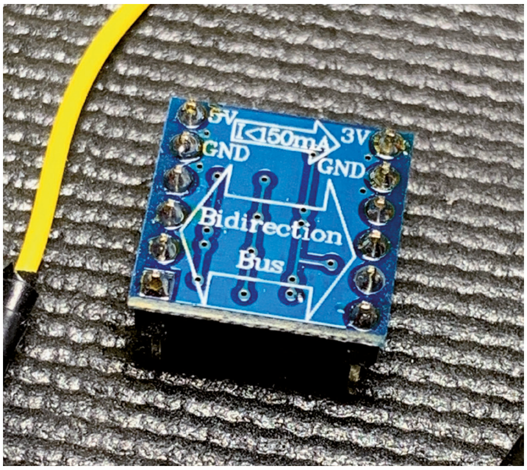


Figure 5-3. The topside of the 3/5 volt level converter PCB.

The level converter must have its ground connections wired to the common ground (they are easy to forget). The +5 volt connection is connected to the ESP32 +5V pin and the +3.3 volt connection wired to the ESP32 +3.3 volt pin. With those connections in place, the bi-directional converter will convert 3.3-volt signals to 5 volts and vice versa. The direction is determined by the end that is driving the signal line. The trigger signal is driven by the ESP32 GPIO 25 on the +3.3 volt side of the converter. Thus the converter translates that signal to 5 volts for the receiving HC-SR04. The Echo signal is driven by the HC-SR04 module on the 5-volt side and is converted down to 3.3 volts for the receiving ESP32.

Using a DMM (Digital Multi Meter), double-check the voltages present on the 3.3-volt side of the conversion module before wiring the signals to the ESP32. To prevent permanent damage, the ESP32 must never see 5-volt signals.

```
// hcsr04.ino
// MIT License (see file LICENSE)

// Set to zero if not using OLED
#define USE_SSD1306   1

// GPIO definitions:
// LED is active high
#define GPIO_LED      12
#define GPIO_TRIGGER  25
#define GPIO_ECHO     26

#if USE_SSD1306
#include "SSD1306.h"

#define SSD1306_ADDR  0x3C
#define SSD1306_SDA   5
#define SSD1306_SCL   4

static SSD1306 oled(
  SSD1306_ADDR,
  SSD1306_SDA,
  SSD1306_SCL
);
#endif

typedef unsigned long usec_t;

static SemaphoreHandle_t barrier;
static TickType_t repeat_ticks = 1000;

//
// Report the distance in CM
//
static void report_cm(usec_t usecs) {
  unsigned cm, tenths;

  cm = usecs * 10ul / 58ul;
  tenths = cm % 10;
  cm /= 10;

  printf("Distance %u.%u cm, %u usecs\n",
    cm,tenths,usecs);

#if USE_SSD1306
  {
    char buf[40];

    snprintf(buf,sizeof buf,
      "%u.%u cm",
      cm,tenths);
    oled.setColor(BLACK);
    oled.fillRect(0,27,128,64);
    oled.setColor(WHITE);
    oled.drawString(64,35,buf);
    oled.display();
  }
#endif
}

//
// Range Finder Task:
//
static void range_task(void *argp) {
  BaseType_t rc;
  usec_t usecs;

  for (;;) {
    rc = xSemaphoreTake(barrier,portMAX_DELAY);
    assert(rc == pdPASS);

    // Send ping:
    digitalWrite(GPIO_LED,HIGH);
    digitalWrite(GPIO_TRIGGER,HIGH);
    delayMicroseconds(10);
    digitalWrite(GPIO_TRIGGER,LOW);

    // Listen for echo:
    usecs = pulseInLong(GPIO_ECHO,HIGH,50000);
    digitalWrite(GPIO_LED,LOW);

    if ( usecs > 0 && usecs < 50000UL )
      report_cm(usecs);
    else
      printf("No echo\n");
  }
}

//
// Send sync to range_task every 1 sec
//
static void sync_task(void *argp) {
  BaseType_t rc;
  TickType_t ticktime;

  delay(1000);

  ticktime = xTaskGetTickCount();

  for (;;) {
    vTaskDelayUntil(&ticktime,repeat_ticks);
    rc = xSemaphoreGive(barrier);
    // assert(rc == pdPASS);
  }
}

//
// Program Initialization
//
void setup() {
  int app_cpu = xPortGetCoreID();
  TaskHandle_t h;
  BaseType_t rc;

  barrier = xSemaphoreCreateBinary();
  assert(barrier);

  pinMode(GPIO_LED,OUTPUT);
  digitalWrite(GPIO_LED,LOW);
  pinMode(GPIO_TRIGGER,OUTPUT);
  digitalWrite(GPIO_TRIGGER,LOW);
  pinMode(GPIO_ECHO,INPUT_PULLUP);

#if USE_SSD1306
  oled.init();
  oled.clear();
  oled.flipScreenVertically();
  oled.setColor(WHITE);
  oled.setTextAlignment(TEXT_ALIGN_CENTER);
  oled.setFont(ArialMT_Plain_24);
  oled.drawString(64,0,"hcsr04.ino");
  oled.drawHorizontalLine(0,0,128);
  oled.drawHorizontalLine(0,26,128);
  oled.display();
#endif

  delay(2000); // Allow USB to connect

  printf("\nhcsr04.ino:\n");

  rc = xTaskCreatePinnedToCore(
    range_task,
    "rangetsk",
    2048,     // Stack size
    nullptr,
    1,        // Priority
    &h,       // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
  assert(h);

  rc = xTaskCreatePinnedToCore(
    sync_task,
    "synctsk",
    2048,     // Stack size
    nullptr,
    1,        // Priority
    &h,       // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
  assert(h);
}

// Not used:
void loop() {
  vTaskDelete(nullptr);
}
```

Listing 5-1. Binary semaphore demo program hsr04.ino.

Figure 5-4 illustrates the breadboard setup used for testing (using the builtin OLED). If you're not using the OLED, the same information can be viewed in the Arduino Serial Monitor output. An example of the Serial Monitor output is shown here:

```
hcsr04.ino:
Distance 125.1 cm, 7257 usecs
Distance 95.1 cm, 5516 usecs
Distance 15.2 cm, 887 usecs
Distance 13.2 cm, 766 usecs
Distance 11.3 cm, 661 usecs
Distance 7.6 cm, 442 usecs
Distance 7.7 cm, 448 usecs
Distance 7.6 cm, 442 usecs
Distance 7.6 cm, 442 usecs
Distance 11.1 cm, 649 usecs
Distance 14.8 cm, 863 usecs
Distance 21.7 cm, 1263 usecs
```

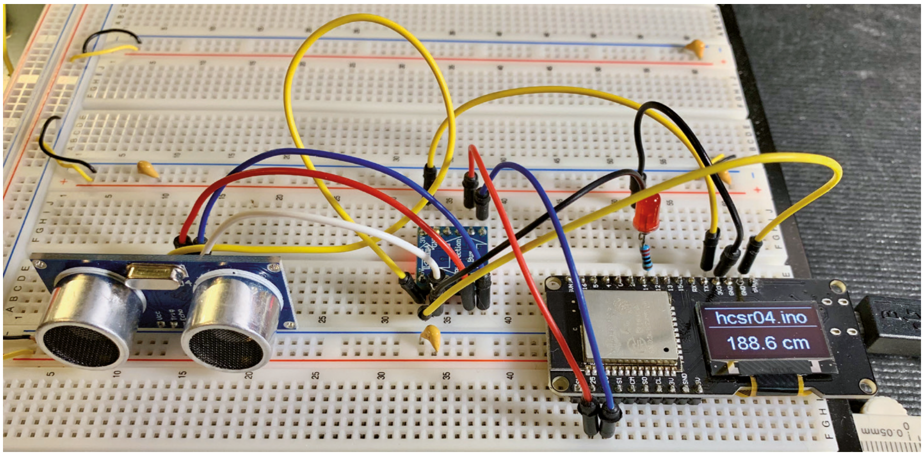


Figure 5-4. A demonstration of the HC-SR04 driven by the ESP32, with voltage level converter pcb in the center.

Program Operation

Let's now review the program's operation. Line 115 of the setup() function creates the binary semaphore and saves the handle in the variable named barrier. Then the GPIO pins are configured and levels established in lines 118 to 122. At the end of the setup() two tasks are created:

1. **range_task** – will drive the HC-SR04 and measure the echo.
2. **sync_task** – will synchronize the range task with timing.

The sync_task() uses the vTaskDelayUntil() function to create accurate one-second timing intervals. All this task needs to do is to "give" the semaphore named "barrier", with each passing second (line 102). Notice that the assertion in line 103 has been commented out. This prevents the assertion from failing if a "take" was not already performed. We simply want to make sure that the range_task is unblocked at that point. Once the program is fully operational, the give operation is always expected to succeed.

Task range_task() makes use of the binary semaphore so that each ping is synchronized to the start of each second. This synchronization is performed by calling xSemaphoreTake() in line 69. Notice that the second parameter supplies the value portMAX_DELAY. Thus the task will not execute past this point until the sync_task() has "given" the semaphore. The semaphore says "None shall pass!" until permission has been given by the sync_task().

Triggering the HC-SR04 requires a minimum pulse of 10 µsec (lines 74 to 76). This provides time enough for the microcontroller to notice the trigger signal is active and queue up a ping for the sending transducer. When the module receives an echo, it will create an echo signal pulse with a matching pulse width for the ESP32.

To measure the pulse width, the Arduino function pulseInLong() is called in line 79. The last parameter provides a 50-millisecond timeout in case the HC-SR04 doesn't respond. This can happen if it doesn't sense an echo.

The mechanism of the binary semaphore in this demonstration is a simple one. One task blocks with a take operation, while the other unblocks it with a give operation. Using this procedure, one task notifies the other.

Locks

The binary semaphore can be used to lock a resource. This is normally performed with a FreeRTOS mutex for reasons covered in a later chapter. But let's introduce the resource locking concept with the binary semaphore.

As discussed earlier, the binary semaphore is created in the empty (not "given") state. To use it as a lock, we need to immediately initialize the semaphore with a give operation after it is created. This makes the lock available for locking. The binary semaphore itself does not lock anything. It is only effective when the program obeys the convention that:

1. Before using the resource, the lock is taken (from the binary semaphore), in effect locking the resource.
2. The resource is then used by the lock taker.
3. The binary semaphore is then unlocked by the lock taker (binary semaphore is given back).

As long as the entire application obeys this convention for the resource, it will only be used by one task at a time.

A simple example use case would be multiple tasks printing to the Serial Monitor. Without locking, the text sent to the monitor would be mixed up with partial lines from the different tasks. To force the printing of complete lines, the convention used would be:

1. Lock the serial monitor's binary semaphore.
2. Print one line of text.
3. Unlock the serial monitor's binary semaphore.

Deadlocks

Locking can be glibly done when there is only one lock involved. But when there are multiple locks involved, it is possible to "deadlock" (also known as the "deadly embrace"). This happens in locking conflicts like this one:

1. Task A locks resource 1.
2. Task B locks resource 2.
3. Task A now tries to lock resource 2 but is blocked because it is already locked by Task B.
4. Task B now tries to lock resource 1 but is blocked because it is already locked by Task A.

Neither task can get past this point because the other task holds a lock that it needs. This becomes increasingly likely when even more locks are involved. Special design care is needed to make these deadlock scenarios impossible.

One effective prevention strategy is to always acquire your locks in the same sequence. If both Task A and B acquire the lock to resource 1 first, then it becomes impossible to deadlock. Whichever task succeeds in locking resource 1 will always be successful in locking resource 2. This strategy does not work for all designs however when more lockable resources are involved.

Another deadlock prevention technique works by limiting how many tasks can begin the locking procedure using the counting semaphore. This is how the Dining Philosopher's problem prevents deadlocks.

Dining Philosophers
The classic problem[3] states that a group of N philosophers sit about a round table eating spaghetti. There are forks provided so that there is one fork between each philosopher (for a total of N forks). Each philosopher has a fork on his/her left and another on the right. Each philosopher quietly thinks for a time but eventually gets hungry. To eat, he/she grabs the left fork and then the fork on the right (eating spaghetti requires two forks). After eating for a while, the forks are returned to their original places and the philosopher resumes thinking.

The problem is that there are N philosophers with N forks. If all philosophers get hungry at the same time, each will grab the left fork first. When they try to grab the right fork, it will already be grabbed by the philosopher on their right. This leads to an impasse where no philosopher can eat. In other words, the system is deadlocked.

One solution to this problem is to limit the number of hungry philosophers. If there are N philosophers, then only N-1 are permitted to get hungry. Leaving one philosopher to think prevents the system from reaching a deadlock. This works because it will always be possible for at least one philosopher to pick up two forks.

Dining Philosophers Demo
To illustrate both the problem and the counting semaphores solution, the program in listing 5-2 is provided.[4] There is nothing to wire beyond plugging in the USB cable and using the Arduino Serial Monitor after flashing.

The program is configured with one statement in line 5:

```
0005: #define PREVENT_DEADLOCK 1
```

When configured with the value 1, the program will use the FreeRTOS counting semaphore to prevent deadlocks. When configured with zero, the program as supplied will deadlock early. Try it out both ways.

The program depends upon a random number generator. The generator is seeded at line 143 with a calculated value. This permits you to reproduce my results exactly. After your initial experiments, if you want to try some truly random runs, comment line 143 out and then uncomment line 142. The call to hallRead() provides a random seed value from the magnetic readings of your ESP32 hall effect sensor.

A queue is created in line 111. This is used by each of the philosopher tasks to send it's status to the main task in the loop() function. Doing this causes the loop() function to be the only task printing to the Serial Monitor to avoid jumbled lines of text.

Each fork is locked by a binary semaphore, which is created in line 115. Because we are using the binary semaphore as a lock (like a mutex), the semaphore is immediately given in line 117. This initializes the semaphore as ready for the taking.

When the macro PREVENT_DEADLOCK is configured as 1, the counting semaphore is created in lines 127 to 131:

```
0127:   csem = xSemaphoreCreateCounting(
0128:     N_EATERS,
0129:     N_EATERS
0130:   );
0131:   assert(csem);
```

The first parameter is the maximum value that the counting semaphore will support. The second parameter is the initial value for the semaphore. In this program, it is set to N_EATERS, which is defined as N-1. This is the number of philosophers that are permitted to simultaneous be hungry.

After the tasks have been created and setup() completed, the philosopher tasks philo_task() execute. Each philosopher's task enters a forever loop starting at line 63. A random delay is used when transitioning between Thinking, Hungry, and Eating states.

When deadlock prevention is enabled, the counting semaphore is "taken" in line 73. If the semaphore is already at the count of zero, the task will be blocked until the semaphore has been given by another task (due to argument portMAX_DELAY). No timeout is used.

When the task succeeds at taking the counting semaphore in line 73, it proceeds to pick up the forks, by using the lock procedure in lines 78 to 84. A random delay is provided at line 82 to make the task more likely to deadlock.

When eating for a random amount of time (lines 86 to 88), the forks are then unlocked (given back) in lines 91 to 95. With deadlock prevention, the counting semaphore is also given back in line 98 to allow another eater.

Deadlock Prevention
Deadlock prevention is configured by:

```
0005: #define PREVENT_DEADLOCK 1
```

When deadlock prevention is enabled, it is provided by the counting semaphore (handle csem). The counting semaphore allows zero to N-1 eaters, but never N. By limiting the number of eaters to 3 (N=4 in this demo), deadlock cannot occur. You should see lines like the following in your Serial Monitor:

```
The Dining Philosopher's Problem:
There are 4 Philosophers.
With deadlock prevention.
00001: Philosopher 1 is Thinking
00002: Philosopher 2 is Thinking
00003: Philosopher 1 is Hungry
00004: Philosopher 0 is Thinking
00005: Philosopher 2 is Hungry
00006: Philosopher 3 is Thinking
00007: Philosopher 3 is Hungry
00008: Philosopher 0 is Hungry
00009: Philosopher 3 is Eating
00010: Philosopher 2 is Eating
00011: Philosopher 1 is Eating
00012: Philosopher 3 is Thinking
00013: Philosopher 3 is Hungry
00014: Philosopher 2 is Thinking
00015: Philosopher 2 is Hungry
00016: Philosopher 1 is Thinking
...
```

Lockups
Deadlock prevention is disabled with the macro setting:

```
0005: #define PREVENT_DEADLOCK 0
```

With deadlock prevention disabled, after compiling and flashing, the demo program locks up immediately when using the random numbers provided. The output should look like this:

```
The Dining Philosopher’s Problem:
There are 4 Philosophers.
Without deadlock prevention.
00001: Philosopher 1 is Thinking
00002: Philosopher 2 is Thinking
00003: Philosopher 1 is Hungry
00004: Philosopher 0 is Thinking
00005: Philosopher 2 is Hungry
00006: Philosopher 3 is Thinking
00007: Philosopher 3 is Hungry
00008: Philosopher 0 is Hungry
(hangs at this point)
```

In this case, all four philosophers get hungry and grab the left fork at almost the same time. After that happens, none of them can grab the right fork and causes the system to lockup. But as an experiment, try changing the setup() by commenting out line 143 and *uncomment* line 142 as shown:

```
0138:   // Initialize for tasks:
0139:   for ( unsigned x=0; x<N; ++x ) {
0140:     philosophers[x].num = x;
0141:     philosophers[x].state = Thinking;
0142:     philosophers[x].seed = hallRead(); // uncommented
0143:     // philosophers[x].seed = 7369+x;
0144:   }
```

You might find that the program appears to work, or at least works some of the time. Lock-ups can drive you mad. They can also be time-consuming to debug.

```
0001: // countsem.ino
0002: // The Dining Philosophers Problem
0003: // MIT License (see file LICENSE)
0004:
0005: #define PREVENT_DEADLOCK 1
0006: #define N                4
0007: #define N_EATERS         (N-1)
0008:
0009: static QueueHandle_t     msgq;
0010: static SemaphoreHandle_t csem;
0011: static int app_cpu = 0;
0012:
0013: enum State {
0014:   Thinking=0,
  Hungry,
  Eating
};

static const char *state_name[] = {
  "Thinking",
  "Hungry",
  "Eating"
};

struct s_philosopher {
  TaskHandle_t  task;
  unsigned      num;
  State         state;
  unsigned      seed;
};

struct s_message {
  unsigned      num;
  State         state;
};

static s_philosopher philosophers[N];
static SemaphoreHandle_t forks[N];
static volatile unsigned logno = 0;

//
// Send the P. state by queue
//
static void send_state(s_philosopher *philo) {
  s_message msg;
  BaseType_t rc;

  msg.num = philo->num;
  msg.state = philo->state;
  rc = xQueueSendToBack(msgq,&msg,portMAX_DELAY);
}

//
// The Philosopher task
//
static void philo_task(void *arg) {
  s_philosopher *philo = (s_philosopher*)arg;
  SemaphoreHandle_t fork1=0, fork2=0;
  BaseType_t rc;

  delay(rand_r(&philo->seed)%5+1);

  for (;;) {
    philo->state = Thinking;
    send_state(philo);
    delay(rand_r(&philo->seed)%5+1);

    philo->state = Hungry;
    send_state(philo);
    delay(rand_r(&philo->seed)%5+1);

#if PREVENT_DEADLOCK
    rc = xSemaphoreTake(csem,portMAX_DELAY);
    assert(rc == pdPASS);
#endif

    // Pick up forks:
    fork1 = forks[philo->num];
    fork2 = forks[(philo->num+1) % N];
    rc = xSemaphoreTake(fork1,portMAX_DELAY);
    assert(rc == pdPASS);
    delay(rand_r(&philo->seed)%5+1);
    rc = xSemaphoreTake(fork2,portMAX_DELAY);
    assert(rc == pdPASS);

    philo->state = Eating;
    send_state(philo);
    delay(rand_r(&philo->seed)%5+1);

    // Put down forks:
    rc = xSemaphoreGive(fork1);
    assert(rc == pdPASS);
    delay(1);
    rc = xSemaphoreGive(fork2);
    assert(rc == pdPASS);

#if PREVENT_DEADLOCK
    rc = xSemaphoreGive(csem);
    assert(rc == pdPASS);
#endif
  }
}

//
// Program Initialization
//
```

```
void setup() {
  BaseType_t rc;

  app_cpu = xPortGetCoreID();
  msgq = xQueueCreate(30,sizeof(s_message));
  assert(msgq);

  for ( unsigned x=0; x<N; ++x ) {
    forks[x] = xSemaphoreCreateBinary();
    assert(forks[x]);
    rc = xSemaphoreGive(forks[x]);
    assert(rc == pdPASS);
    assert(forks[x]);
  }

  delay(2000); // Allow USB to connect
  printf("\nThe Dining Philosopher’s Problem:\n");
  printf("There are %u Philosophers.\n",N);

#if PREVENT_DEADLOCK
  csem = xSemaphoreCreateCounting(
    N_EATERS,
    N_EATERS
  );
  assert(csem);
  printf("With deadlock prevention.\n");
#else
  csem = nullptr;
  printf("Without deadlock prevention.\n");
#endif

  // Initialize for tasks:
  for ( unsigned x=0; x<N; ++x ) {
    philosophers[x].num = x;
    philosophers[x].state = Thinking;
    // philosophers[x].seed = hallRead();
    philosophers[x].seed = 7369+x;
  }

  // Create philosopher tasks:
  for ( unsigned x=0; x<N; ++x ) {
    rc = xTaskCreatePinnedToCore(
      philo_task,
      "philotsk",
      5000,             // Stack size
      &philosophers[x], // Parameters
      1,                // Priority
      &philosophers[x].task, // handle
      app_cpu           // CPU
    );
    assert(rc == pdPASS);
    assert(philosophers[x].task);
  }
}

//
// Report philosopher states:
//
void loop() {
  s_message msg;

  while ( xQueueReceive(msgq,&msg,1) == pdPASS ) {
    printf("%05u: Philosopher %u is %s\n",
      ++logno,
      msg.num,
      state_name[msg.state]);
  }
  delay(1);
}
```

Listing 5-2. The Dining Philosophers program countsem.ino

Insidious Deadlocks
In computer science literature, there is a fair amount of discourse on deadlock prevention and with good reason. Sometimes good design can be satisfied with just the proper sequencing of the locks. The dining philosophers problem was made safe through the use of a counting semaphore. When several resources require simultaneous locking however, the problem can become acute and other methods must be applied.

The nature of the problem is insidious. A project can be tested successfully for weeks. True to Murphy’s law, however, when you demonstrate your tested IoT device live at a Hackaday Superconference, it will lockup. This leads to the following conclusion:

"Testing alone is not proof of deadlock safety. It must be designed to be deadlock safe."

When creating devices that lives depends upon, they need to be designed impossible to deadlock. Even a hobby project can benefit from this design purity. Careful design may prevent your ESP32 project from becoming a "throwie".[6]

Summary

A binary semaphore can be used to synchronize two tasks. While a binary semaphore can also be used for locking a resource, you will later learn why a mutex is a preferred tool for the job. You have seen one application of the counting semaphore where it limited access based upon a maximum count.

The next chapter examines the concept of the FreeRTOS mailbox. As the name implies, this provides another useful communication tool for your independent tasks.

Exercises

1. How does a binary semaphore synchronize tasks?
2. If task A has binary semaphore 1 taken and task B has binary semaphore 2 taken, is there a deadlock?
3. What is another commonly used name for a deadlock?
4. Can a binary semaphore unblock multiple tasks?
5. What is the initial state of a binary semaphore? Empty or given?
6. What is the initial state of a counting semaphore?
7. How does the counting semaphore prevent a deadlock in the dining philosophers problem?
8. Does giving a counting semaphore increase or decrease the count?
9. Which initial counting semaphore value indicates that all resources are initially taken?
10. How many states does the binary semaphore have?

Web Resources

[1] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/hcsr04/hcsr04.ino
[2] http://www.hobbytronics.co.uk/mosfet-voltage-level-converter
[3] https://en.wikipedia.org/wiki/Dining_philosophers_problem
[4] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/countsem/countsem.ino
[5] https://en.wikipedia.org/wiki/Murphy%27s_law
[6] https://hackaday.com/tag/throwie/

Chapter 6 • Mailboxes

0xF7 Elm St.

What is so beneficial about a mailbox? Isn't it the ability to drop off mail at the convenience of the mail carrier? And isn't it also about the convenience of the pickup? Therefore the mailbox is a practical system where the timing of delivery and receiving is decoupled.

Applications often need global variables to share data between different modules. With independently executing tasks, there is a need for coordination of these global values in an atomic fashion. No task wants to use a partially updated global value deposited by another task. The mailbox is the perfect solution for data structures large and small. The timing of the update is decoupled and is atomic.

The Problem

To appreciate the need for mailboxes, let's review one strategy for sharing global data between tasks. Because tasks execute independently and get preempted based upon a timer tick, there is never a guarantee that the preempted task has fully completed its last operation before the next task gets run by the scheduler. Furthermore compiled code uses various optimizations including the keeping of values in a register whenever possible. So even the process of storing a simple integer to a global variable is potentially problematic when sharing between tasks.

The C/C++ volatile keyword informs the compiler to not optimize its variable access with register storage. So if the global variable is declared as:

```
        volatile int global_int = 0;
```

and your code performs the following:

```
        global_int = 95;
```

then the compiler forces the compiled code to store the value in memory immediately. Otherwise, the optimized code might keep the value in a register and store it at a later point. Likewise, when another task attempts to reference the value, for example:

```
        if ( global_int == 42 ) {
            ...
```

the compiled code when volatile is used will be forced to fetch the value global_int from memory when it might otherwise use a value saved in a register.

Without the volatile attribute, the compiler might keep the updated value in a register in the updating task. The task examining that global value might assume that a register still represents the current value of global_int and use that instead. The magic of volatile is that it forces the compiler to perform the store/load memory without using register optimizations. Before you celebrate with glee for that solution, note that there's a catch – the atomicity of the operation depends upon the value's size. The ESP32 is a 32-bit device, so a store to memory of an 8-bit to 32-bit sized value can be performed in a single instruction. So a tick interrupt cannot make that store to memory a partial one. The store will be all or nothing (i.e. atomic). Larger values on the other hand, like structures, require more than one instruction to complete. A tick interrupt occurring at the right time can, therefore, leave that global partially updated.

There is also the multiple CPU problem. While the ESP32 might be able to coordinate memory stores between the dual CPUs, not all architectures guarantee it. Adding data memory cache to the hardware results in the need to coordinate between each cache when executing code on different CPUs.

The designers of FreeRTOS recognized these problems and devised a solution that includes larger objects like structures.

The Mailbox

You have already read about the FreeRTOS queue, so you know how data items can be queued and received atomically. To build a mailbox from a queue, why not simply configure the queue with size set to 1? This almost works but to update the mailbox value, it needs to be empty to re-add the replacement item. To remedy that, FreeRTOS provides the xQueueOverwrite() function.

```
        BaseType_t xQueueOverwrite(
          QueueHandle_t handle,  // Queue handle
          const void *pitem      // Pointer to data item
        );
```

The returned value can only be pdPASS, since this operation always succeeds. The xQueueOverwrite() function is intended to be used in queues with a length of one. When the queue is empty, the item is added and otherwise overwritten.

Creating a Mailbox

The creation of the mailbox is just like the queue, except for the fact that the queue depth is one:

```
QueueHandle_t qh;

qh = xQueueCreate(1,sizeof(my_data_type_t));
assert(qh);
```

One special consideration of the FreeRTOS mailbox is that it is created empty (just like a queue). If you want it to immediately have a value, you should immediately call xQueue-Overwrite() to set it.

Reading the Mailbox

The function xQueueReceive() could be used to read the mailbox. But this call removes the data item out of the queue and leaves the mailbox (queue) empty. If that is what you intended, that is ok. But to maintain a non-empty mailbox, use xQueuePeek() instead:

```
BaseType_t xQueuePeek(
  QueueHandle_t xQueue,
  void *pvBuffer,
  TickType_t xTicksToWait
);
```

Because the mailbox can be potentially empty, you must supply the third argument xTick-sToWait. If your mailbox is never expected to be empty, you can supply zero. Then, if the mailbox is unexpectedly empty, you can catch it with an assertion or error check:

```
BaseType_t rc;
my_data_type_t item;

rc = xQueuePeek(handle,&item,0);
assert(rc == pdPASS);
```

It is perfectly legal to make use of empty/not-empty mailboxes, if your design benefits. This treatment likens it to a global value that is null or otherwise has a value.

Mailbox Demonstration

To provide a meaningful demonstration, this project involves reading from two I2C modules and then reports them to the Serial Monitor. [1]

1. The Si7021 sensor module provides temperature and relative humidity readings.
2. The HMC5883L sensor provides magnetic compass readings.

There are three tasks used in this application:

1. The Si7021 sensor reading task, posting the readings to a mailbox.
2. The HMC5883L sensor reading task, posting the readings to another mailbox.
3. The display task, reporting to the Serial Monitor from the mailboxes.

At first blush, this sounds like a walk in the park. Realize, however, that both sensor tasks must share the same I2C bus. Thus this requires the use of a binary semaphore to prevent both tasks from trying to use the bus simultaneously. The display task must also be informed when the mailbox has updates. A binary semaphore is used for notification.

For those wishing to use the Lolin module's OLED display, be aware that there are additional challenges. The OLED also uses the I2C bus but the library sets the clock to 700 kHz for faster access. This is incompatible with the HMC5883L and Si7021 sensors, which are limited to 400 kHz. For this reason, the demonstration does not include OLED support.

This demonstration program uses the Adafruit Si7021 Library. The version tested was 1.2.3. This is easily installed with the Arduino IDE in the Tools -> Manage Libraries dialog. The program reads temperature and relative humidity readings from the Si7021 sensor and places those into a mailbox. The HMC5883L module is queried and its magnetic readings are placed in another mailbox for the compass. When either of these mailboxes are updated, the semaphore for the display task is notified by the binary semaphore. This wakes it up so that new values will be displayed on the Serial Monitor.

Figure 6-1 illustrates the schematic used for this demo and Figure 6-2 illustrates the breadboard setup used. The sensor modules are both powered by +3.3 volts from the ESP32 dev board and require no more than about 100 µA total.

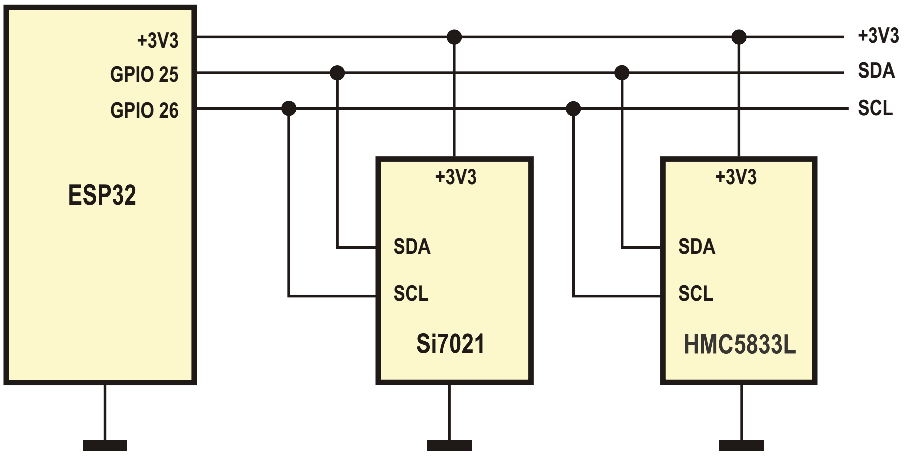


Figure 6-1. The schematic for the demonstration mailbox.ino program.

When the program runs, the serial monitor output should look similar to the following:

```
mailbox.ino:
Temperature:      24.23C, RH 29.95 %
Compass reading not available.
Temperature:      24.23C, RH 29.95 %
Compass readings: 18846, 40606, 27806
Temperature:      24.20C, RH 29.95 %
Compass readings: 18846, 40606, 27806
Temperature:      24.20C, RH 29.95 %
Compass readings: 18846, 40606, 27806
Temperature:      24.19C, RH 29.93 %
Compass readings: 18846, 40606, 27806
Temperature:      24.19C, RH 29.93 %
Compass readings: 18838, 38550, 54678
Temperature:      24.20C, RH 29.94 %
Compass readings: 18838, 38550, 54678
```

The first compass readings may not be immediately available as was the case in this example. This was fortunate because it provided proof of the empty mailbox capability. The temperature and relative humidity readings were in agreement with my health monitor close by. A photo of it is shown in Figure 6-3.

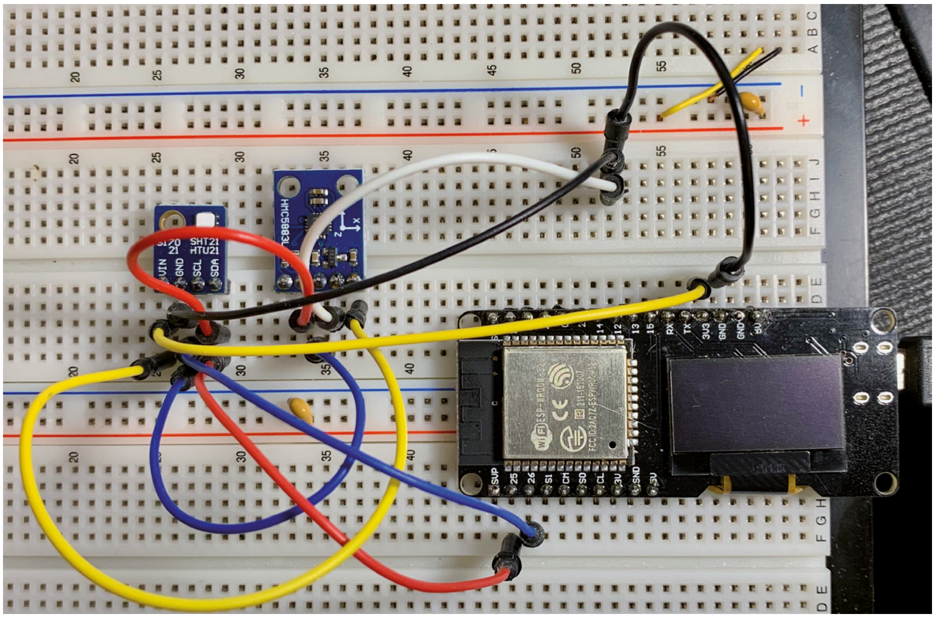


Figure 6.2. A photo of the mailbox.ino program's breadboard setup and I2C sensors.

Figure 6-3. A photo of the nearby health monitor, where the readings were in close agreement.

Program Dissection

Turning our attention to the program internals, let's examine some mailbox related code. Each of the two mailboxes used in this program are structures:

```
struct s_compass {
  uint16_t  x;
  uint16_t  y;
  uint16_t  z;
};

struct s_temp {
  float     temp;
  float     humidity;
};
```

The compass sensor returns x, y, and z values as uint16_t values in the s_compass structure. These can be converted into a degree heading with some calibration but this was omitted for demo simplicity. The temperature and humidity readings are saved as float value members in the s_temp structure.

To create the corresponding mailboxes, the following code was used in the setup() function:

```
  // Compass Mailbox:
  comph = xQueueCreate(1,sizeof(s_compass));
  assert(comph);

  // Temperature and RH Mailbox:
  temph = xQueueCreate(1,sizeof(s_temp));
  assert(temph);
```

Each mailbox is created with a queue depth of one, with the data item size set to the size of the structure. In this demonstration, we permitted the mailboxes to remain empty until they were populated. The display task anticipates this possibility.

Because we haven't covered mutexes yet, a binary semaphore was used instead to lock the I2C bus:

```
0202:   // I2C locking semaphore:
0203:   i2sem = xSemaphoreCreateBinary();
0204:   assert(i2sem);
0205:   rc = xSemaphoreGive(i2sem);
0206:   assert(rc == pdPASS);
```

Recall that a binary semaphore is created empty, so it must be initialized as given to act as a locking semaphore.

The main loop of the temperature and humidity sensor loop, is representative of both sensor tasks, even though the details of reading vary:

```
0056:   s_temp reading;
...
0073:   for (;;) {
0074:     i2c_lock();
0075:     reading.temp = si7021.readTemperature();
0076:     reading.humidity = si7021.readHumidity();
0077:     i2c_unlock();
0078:
0079:     rc = xQueueOverwrite(temph,&reading);
0080:     assert(rc == pdPASS);
0081:
0082:     // Notify disp_task:
0083:     xSemaphoreGive(chsem);
0084:
0085:     delay(500);
0086:   }
```

Lines 74 and 77 lock and unlock the I2C bus respectively, to prevent collision of the two sensor reading tasks on the same I2C bus. The structure s_temp is populated with the readings from lines 75 and 76. The structure is then written to the mailbox in line 79. FreeRTOS performs this operation in an atomic manner.

If nothing else was performed, the mailbox would be updated but our display task wouldn't be aware of it. Line 83 performs a give operation on the change semaphore (chsem). This gives a shove to the display task, causing it to display new values.

The display task is blocked by the change semaphore at the top of its loop until it is notified:

```
0158:   s_temp temp_reading;
...
0162:   for (;;) {
0163:     // Wait for change notification:
0164:     rc = xSemaphoreTake(chsem,portMAX_DELAY);
0165:     assert(rc == pdPASS);
```

After becoming unblocked, the display task then checks both mailboxes. The following is representative of both:

```
0167:     // Grab temperature, if any:
0168:     rc = xQueuePeek(temph,&temp_reading,0);
0169:     if ( rc == pdPASS ) {
0170:       printf("Temperature:      %.2fC, RH %.2f %%\n",
0171:         temp_reading.temp,
0172:         temp_reading.humidity);
0173:     } else {
0174:       printf("Temperature & RH not available.\n");
0175:     }
```

The xQueuePeek() call prevents the mailbox from becoming emptied while copying its current structure values to the local structure in line 158. If the peek was successful, then the readings are reported to the Serial Monitor. If the mailbox is empty, then a message that the reading is not available is reported instead. The same operation is performed for the compass readings.

```
0001: // mailbox.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // Configuration
0005: #define I2C_SDA       25
0006: #define I2C_SCL       26
0007: #define DEV_Si7021    (uint8_t(0x1E))
0008: #define DEV_HMC5883L  (uint8_t(0x40))
0009:
0010: #include <Wire.h>
0011: #include <Adafruit_Si7021.h>
0012:
0013: static Adafruit_Si7021 si7021;
0014:
0015: static int app_cpu = 0;
0016: static QueueHandle_t comph = nullptr;
0017: static QueueHandle_t temph = nullptr;
0018: static SemaphoreHandle_t chsem = nullptr;
```

```
static SemaphoreHandle_t i2sem = nullptr;

struct s_compass {
  uint16_t  x;
  uint16_t  y;
  uint16_t  z;
};

struct s_temp {
  float     temp;
  float     humidity;
};

//
// Lock I2C Bus
//
static inline void i2c_lock() {
  BaseType_t rc;

  rc =  xSemaphoreTake(i2sem,portMAX_DELAY);
  assert(rc == pdPASS);
}

//
// Unlock I2C Bus
//
static inline void i2c_unlock() {
  BaseType_t rc;

  rc = xSemaphoreGive(i2sem);
  assert(rc == pdPASS);
}

//
// Temperature and Humidity Task
//
static void temp_task(void *argp) {
  s_temp reading;
  uint8_t er;
  BaseType_t rc;

  i2c_lock();

  if ( !si7021.begin() ) {
    i2c_unlock();
    vTaskDelete(nullptr);
  }

  si7021.reset();
  i2c_unlock();

  reading.temp = 0.0;
  reading.humidity = 0.0;

  for (;;) {
    i2c_lock();
    reading.temp = si7021.readTemperature();
    reading.humidity = si7021.readHumidity();
    i2c_unlock();

    rc = xQueueOverwrite(temph,&reading);
    assert(rc == pdPASS);

    // Notify disp_task:
    xSemaphoreGive(chsem);

    delay(500);
  }
}

//
// Read MSB + LSB for compass reading
//
static inline int16_t read_i16() {
  uint8_t ub1, ub2;

  ub1 = Wire.read();
  ub2 = Wire.read();
  return int16_t((ub1 << 8)|ub2);
}

//
// Compass reading task:
//
static void comp_task(void *argp) {
  s_compass reading;
  BaseType_t rc;
  int16_t i16;
  uint8_t s;
  bool status;

  for (;;) {
    status = false;

    i2c_lock();
    Wire.beginTransmission(DEV_HMC5883L);
    Wire.write(9);  // Status register
    rc = Wire.requestFrom(DEV_HMC5883L,uint8_t(1));
    if ( rc != 1 ) {
      i2c_unlock();
      printf("I2C Fail for HMC5883L\n");
      sleep(1000);
      continue;
    }

    s = Wire.read();  // Status
    i2c_unlock();

    if ( !(s & 0x01) )
      continue;

    // Device is ready for reading
    i2c_lock();
    Wire.beginTransmission(DEV_HMC5883L);
    Wire.write(3);
    rc = Wire.requestFrom(DEV_HMC5883L,uint8_t(6));
    if ( rc == 6 ) {
      reading.x = read_i16();
      reading.z = read_i16();
      reading.y = read_i16();
      status = true;
    }
    i2c_unlock();

    if ( status ) {
      rc = xQueueOverwrite(comph,&reading);
      assert(rc == pdPASS);

      // Notify disp_task:
      xSemaphoreGive(chsem);
    }
    delay(500);
  }
}

//
// Display task (Serial Monitor)
//
```

```
static void disp_task(void *argp) {
  s_temp temp_reading;
  s_compass comp_reading;
  BaseType_t rc;

  for (;;) {
    // Wait for change notification:
    rc = xSemaphoreTake(chsem,portMAX_DELAY);
    assert(rc == pdPASS);

    // Grab temperature, if any:
    rc = xQueuePeek(temph,&temp_reading,0);
    if ( rc == pdPASS ) {
      printf("Temperature:      %.2fC, RH %.2f %%\n",
        temp_reading.temp,
        temp_reading.humidity);
    } else {
      printf("Temperature & RH not available.\n");
    }

    // Grab compass readings, if any:
    rc = xQueuePeek(comph,&comp_reading,0);
    if ( rc == pdPASS ) {
      printf("Compass readings: %d, %d, %d\n",
        comp_reading.x,
        comp_reading.y,
        comp_reading.z);
    } else {
      printf("Compass reading not available.\n");
    }
  }
}

//
// Program Initialization
//
void setup() {
  BaseType_t rc;

  app_cpu = xPortGetCoreID();

  // Change notification:
  chsem = xSemaphoreCreateBinary();
  assert(chsem);

  // I2C locking semaphore:
```

```
  i2sem = xSemaphoreCreateBinary();
  assert(i2sem);
  rc = xSemaphoreGive(i2sem);
  assert(rc == pdPASS);

  // Compass Mailbox:
  comph = xQueueCreate(1,sizeof(s_compass));
  assert(comph);

  // Temperature and RH Mailbox:
  temph = xQueueCreate(1,sizeof(s_temp));
  assert(temph);

  // Start I2C Bus Support:
  Wire.begin(I2C_SDA,I2C_SCL);

  // Allow USB to Serial to start:
  delay(2000);
  printf("\nmailbox.ino:\n");

  // Temperature Reading Task:
  rc = xTaskCreatePinnedToCore(
    temp_task,
    "temptsk",
    2400,     // Stack size
    nullptr,
    1,        // Priority
    nullptr,  // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);

  // Compass Reading Task:
  rc = xTaskCreatePinnedToCore(
    comp_task,
    "comptsk",
    2400,     // Stack size
    nullptr,
    1,        // Priority
    nullptr,  // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);

  // Display task:
  rc = xTaskCreatePinnedToCore(
    disp_task,
    "disptsk",
    4000,     // Stack size
    nullptr,
    1,        // Priority
    nullptr,  // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
}

// Not used:
void loop() {
  vTaskDelete(nullptr);
}
```

Listing 6-1. The mailbox.ino demonstration program.

Summary

This chapter demonstrated that the mailbox is a useful and special case of the queue. The FreeRTOS mailbox provides atomic capability for structured items. The demo program presented used initially empty mailboxes. Your programs can initialize the mailboxes instead, if that is the requirement. The queue overwrite and queue peek operations were used on the mailboxes instead of the normal queue send and receive operations.

Exercises

1. Which API function is used to fetch from a mailbox?
2. What is the initial state of the mailbox after it is created?
3. Why is the function xQueueOverwrite() used instead of xQueueSendToBack()?
4. Where is the size of the mailbox data item specified?
5. Can the xQueueOverwrite() function block?
6. Why might the call to xQueuePeek() block?
7. When might an empty mailbox be advantageous to an application?

Web Links

[1] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/mailbox/mailbox.ino

Chapter 7 • Task Priorities

Me first!

Some might regard the topic of task priority with a certain degree of ambivalence. The exposure to Linux and Windows systems may cause them to wonder what the hoopla is all about. Under traditional operating systems everything executes, albeit with some processes faster than others. This is the popular notion of execution priority.

FreeRTOS task priority, on the other hand, can be thought of as "hardcore priority". If task A has a higher priority than task B, then task B will never execute when task A is always ready to run. This naturally has design consequences. FreeRTOS was designed from the start to respect priority at every turn. For this reason, the hobbyist and engineer alike need to be clear about the way that task priorities operate in FreeRTOS.

vTaskStartScheduler()

Somewhere within the executable program, the FreeRTOS function vTaskStartScheduler() must be called. *Be aware that the Arduino environment startup does this for you*. It is a fatal error to call it a second time. When working in a non-Arduino environment however, you may be required to call it.

What does vTaskStartScheduler() do?

- It starts the FreeRTOS scheduler.
- Automatically creates the "idle task".
- Automatically creates the "timer daemon task".

It is possible to create tasks before starting the scheduler. The highest priority task will automatically begin when vTaskStartScheduler() is called. You may encounter this in non-Arduino embedded code.

When there are no tasks ready to execute, the scheduler executes the idle task.

Configured Scheduling Algorithm

FreeRTOS can be configured to support modified forms of scheduling. This chapter will assume the configuration used by Arduino. This means that the FreeRTOSConfig.h file defines the macros:

- configUSE_PREEMPTION is true
- configUSE_TIME_SLICING is true
- configUSE_TICKLESS_IDLE is false

The consequence of this configuration is that we'll be working with preemptive scheduling, using time slices affected by system timer ticks. This is the scheduling configuration used by most small RTOS applications but be aware that other configurations are possible.

Task Pre-emption
Let's clarify the meaning of pre-emption as it applies to scheduling. In FreeRTOS, a task running at a given priority is "pre-empted" when another task with *higher* priority enters the Ready state. In other words, the higher priority task immediately begins to execute while the lower priority task is put on hold. This also happens when the other task has its priority increased above the current task's priority. The pre-empted (on hold) task remains in the Ready state but is left waiting for its chance to resume. Unlike cooperative multitasking, this happens involuntarily.

Time Slicing
This is a concept that varies according to the operating system. Some platforms assign a fixed time slice duration to each task. In this way, complete CPU time fairness is achieved.

Within FreeRTOS however, time slicing occurs with the arrival of the system tick interrupt. If task1 starts immediately after the last system tick and then blocks (or yields), the remaining time slice is given to the next task (task2). Task 2 thus does not get a full-time slice because the next system tick will give the CPU to yet another task. The bottom line is that FreeRTOS uses the system tick interrupt to cause the scheduler to run briefly to choose the next eligible task, regardless of the time used in the current slice.

ESP32 Task Priorities
Every programmer wants to know what the limitations of his resources are. For example, how many priority levels are there to choose from? Which represents the lowest urgency – a high numbered priority or a low numbered level?

The ESP32 environment for Arduino defines 25 priority levels, ranging from zero to 24. Priority zero represents the lowest urgency while 24 is the most urgent. By default, the Arduino setup() and loop() functions run at priority level 1 (these functions are part of the same task named loopTask).

Tasks of equal priority are scheduled in a round-robin fashion. At the next tick interrupt, the next equal priority task is scheduled to run, if any. If there are no other ready tasks at the same priority, the scheduler continues to run the current task.

Task States
We're already in trouble with this discussion. What is meant by a ready task? What other task states are there that influence the scheduler? Figure 7-1 illustrates a simplified state diagram for FreeRTOS. The different states that the task can have include:

- Ready – the initial state of a created task.
- Running – the execution state of a task.
- Blocked – a task that is not scheduled because it is waiting for some event.
- Suspended – a task that has been suspended.

When the task is first created by xTaskCreate(), it is immediately created in the Ready state. If the created task has a higher priority than the task that is creating it, the caller is immediately moved from the Running state to the Ready state. The created task then begins execution.

A task in the Running state can eventually block on a queue or semaphore, for example. This causes the task state to change from Running to Blocked. It will stay blocked until the event unblocks, or the specified timeout occurs. The task then changes from Blocked to the Ready state. A task in the Ready state becomes eligible to run again.

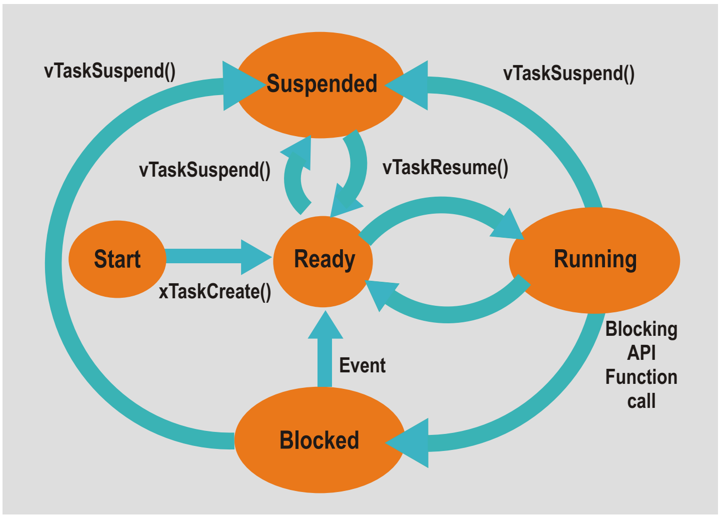


Figure 7-1. The states of a FreeRTOS task.

A task can also be suspended by calling vTaskSuspend(). This can move the task from any current state to the Suspended state. That task will remain suspended until vTaskResume() is called. At that point, a task may go through some substates that are not shown in the diagram. For example, a task blocked on a queue and was then suspended will initially move to the Ready state. Once the scheduler moves the task back to the Running state (to retry the queue operation), the task may immediately return to the Blocked state, unless the queue has changed enough to allow the queue operation to succeed.

I/O and Sharing the CPU

In microcomputer programming, a program must sometimes wait for an I/O peripheral event to occur. The device may be busy and require that the program wait until the device is ready. A simplistic approach would be to use a busy-wait loop:

```
// Wait for device ready:
while ( is_busy() )
    ; // spin
// device is ready
```

This wastes the CPU execution time while the device is busy. If you have other tasks that could benefit from being run, wouldn't it be nice if you could share the CPU? Calling the FreeRTOS function (macro) taskYIELD() invokes the scheduler directly. If other tasks are running at the same priority, they will be scheduled round-robin. If there are none, control returns to the caller:

```
// Wait for device ready:
while ( is_busy() )
    taskYIELD(); // share CPU
// device is ready
```

This is the multitasking friendly way to make the most effective use of your CPU resource.

Preventing Immediate Task Start

It is not always desirable for a task to start immediately when it is created. Created tasks will start immediately unless otherwise restrained. So how do you create a task that does not start until you want it to?

One approach is to use a binary semaphore or some other event mechanism. But this requires additional resources and seems clumsy. The one factor in your immediate control is the task priority that is specified for the task creation. If the calling task has priority 2, then creating the task at priority 1 or lower will prevent it from executing, as long as the current task does not block or yield. Once the new task has been created, you can change the new task's priority with vTaskPrioritySet() when you want it to start. But be careful with this approach – if the creating task blocks for any reason, the scheduler is permitted to run a lower priority task.

Note: When creating tasks that should not begin immediately, create them with a priority lower than the current task. Release the created task later by changing its priority using vTaskPrioritySet(). Be careful that the current task does not block for any reason because this allows lower priority tasks to execute.

Simple Demonstration

A simple demonstration of how priority affects the task startup is provided in Listing 7-1.[1] The program begins using the Arduino loopTask function setup(). First the setup() routine changes the priority of the loopTask to 3 and proves it by reporting it on the Serial Monitor:

```
vTaskPrioritySet(nullptr,3);
priority = uxTaskPriorityGet(nullptr);
assert(priority == 3);

```

```
0047:   printf("\ntaskcreate.ino:\n");
0048:   printf("loopTask priority is %u.\n",
0049:     priority);
```

Once that is out of the way, the task named task1 is created at priority 2 and then announced immediately after:

```
0051:   rc = xTaskCreatePinnedToCore(
0052:     task1,
0053:     "task1",
0054:     2000,     // Stack size
0055:     nullptr,
0056:     2,        // Priority
0057:     &h1,      // Task handle
0058:     app_cpu   // CPU
0059:   );
0060:   assert(rc == pdPASS);
0061:   // delay(1);
0062:   printf("Task1 created.\n");
```

Ignore commented line 61 for now. The task is created at priority 2, which is lower than the current task's priority of 3. Task1 also makes its own announcement when it starts:

```
0017:   printf("Task1 executing, priority %u.\n",
0018:     (unsigned)uxTaskPriorityGet(nullptr));
```

by printing its priority and announcing itself in lines 17 and 18. When you flash and run the program, you get the following Serial Monitor output:

```
taskcreate.ino:
loopTask priority is 3.
Task1 created.
Task1 executing, priority 3.
```

Pay attention to the sequence of messages displayed. The line:

```
Task1 created.
```

is reported by line 62 of the creating setup() function. This is reported ahead of the line:

```
Task1 executing, priority 3.
```

which is reported by lines 17 and 18 of task1. This demonstrates that the lower priority of task1 (concerning the setup() function of the loopTask) caused it to wait until the loopTask raised task1's priority to 3 in line 64:

```
0064:   vTaskPrioritySet(h1,3);
```

This determined the sequence of execution.

Task1 then creates another task at priority 4:

```
0020:   rc = xTaskCreatePinnedToCore(
0021:     task2,
0022:     "task2",
0023:     2000,     // Stack size
0024:     nullptr,
0025:     4,        // Priority
0026:     nullptr,  // Task handle
0027:     app_cpu   // CPU
0028:   );
0029:   assert(rc == pdPASS);
0030:   printf("Task2 created.\n");
```

What does the Serial Monitor show?

```
taskcreate.ino:
loopTask priority is 3.
Task1 created.
Task1 executing, priority 3.
Task2 executing, priority 4.
Task2 created.
```

From this we see that the "Task2 created." message follows task2's message "Task 2 executing, priority 4". Clearly, task2 began executing before the task create call in line 20 returned.

Blocking

What happens if the creating task blocks for some reason, even though the created task has a lower priority? Uncomment line 61 and run the program again:

```
0061:   delay(1);  // uncomment
```

When the program is flashed and run again, the Serial Monitor output appears as follows:

```
taskcreate.ino:
loopTask priority is 3.
Task1 executing, priority 2.
Task2 executing, priority 4.
Task2 created.
Task1 created.
```

What happened? We see that as soon as the call to delay occurs (which blocks the loopTask for a short time), task1 executes and prints its banner message "Task1 executing". The setup() function's print of "Task1 created." is now the last line reported because the created task2 has higher priority also.

```
// taskcreate.ino
// MIT License (see file LICENSE)

static int app_cpu = 0;

void task2(void *argp) {

  printf("Task2 executing, priority %u.\n",
    (unsigned)uxTaskPriorityGet(nullptr));
  vTaskDelete(nullptr);
}

void task1(void *argp) {
  BaseType_t rc;
  TaskHandle_t h2;

  printf("Task1 executing, priority %u.\n",
    (unsigned)uxTaskPriorityGet(nullptr));

  rc = xTaskCreatePinnedToCore(
    task2,
    "task2",
    2000,     // Stack size
    nullptr,
    4,        // Priority
    nullptr,  // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
  printf("Task2 created.\n");
  vTaskDelete(nullptr);
}

void setup() {
  BaseType_t rc;
  unsigned priority = 0;
  TaskHandle_t h1;

  app_cpu = xPortGetCoreID();

  delay(2000); // Allow USB init time

  vTaskPrioritySet(nullptr,3);
  priority = uxTaskPriorityGet(nullptr);
  assert(priority == 3);

  printf("\ntaskcreate.ino:\n");
  printf("loopTask priority is %u.\n",
    priority);

  rc = xTaskCreatePinnedToCore(
    task1,
    "task1",
    2000,     // Stack size
    nullptr,
    2,        // Priority
    &h1,      // Task handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
  // delay(1);
  printf("Task1 created.\n");

  vTaskPrioritySet(h1,3);
}

// Not used:
void loop() {
  vTaskDelete(nullptr);
}
```

Listing 7-1. Program taskcreate.ino demonstrating how task priority affects created tasks.

Creating a Ready-to-Go Task

Perhaps you'd rather be able to create a task that is all ready to go, with its priority pre-configured. This can be done by suspending the created task. Listing 7-2 demonstrates the following procedure[2]:

1. The calling task loopTask runs at priority 1 (by default)
2. Creates the new task named task1, with priority zero (lines 24 to 33).
3. Suspends the new task (line 38).
4. Configures the required priority for the new task (line 39).
5. When ready, launches the fully configured task by resuming it (line 44).

When the demonstration is flashed and run, the following should be seen on the Serial Monitor:

```
taskcreate2.ino:
loopTask priority is 1.
Task1 created.
Zzzz... 3 secs
Task1 executing, priority 3.
```

This session demonstrates the order of events. The loopTask runs at its default priority of 1, while the created task gets created at priority zero. Because of this, the loopTask continues to execute and can suspend the new task and then configure its priority to 3. The suspended task does not run until it is resumed by the call to vTaskResume() in line 44.

```
0001: // taskcreate2.ino
0002: // MIT License (see file LICENSE)
0003:
0004: static int app_cpu = 0;
0005:
0006: void task1(void *argp) {
0007:
0008:   printf("Task1 executing, priority %u.\n",
0009:     (unsigned)uxTaskPriorityGet(nullptr));
0010:   vTaskDelete(nullptr);
0011: }
0012:
0013: void setup() {
0014:   BaseType_t rc;
0015:   TaskHandle_t h1;
0016:
0017:   app_cpu = xPortGetCoreID();
0018:
0019:   delay(2000); // Allow USB init time
0020:
0021:   printf("\ntaskcreate2.ino:\n");
0022:   printf("loopTask priority is %u.\n",
0023:     (unsigned)uxTaskPriorityGet(nullptr));
0024:
0025:   rc = xTaskCreatePinnedToCore(
0026:     task1,
0027:     "task1",
0028:     2000,     // Stack size
0029:     nullptr,
0030:     0,        // Priority
0031:     &h1,      // Task handle
0032:     app_cpu   // CPU
```

```
  );
  assert(rc == pdPASS);

  printf("Task1 created.\n");

  vTaskSuspend(h1);
  vTaskPrioritySet(h1,3);

  printf("Zzzz... 3 secs\n");
  delay(3000);

  vTaskResume(h1);
}

// Not used:
void loop() {
  vTaskDelete(nullptr);
}
```

Listing 7-2. Creating a task, setting suspending and setting priority

ESP32 Dual Core Wrinkle
FreeRTOS was originally developed for single-core CPU microcontrollers. Because the ESP32 consists of a dual CPU arrangement, Espressif customized the scheduler component of FreeRTOS. As a review, the following ESP32 CPUs are present:

1. CPU 0 known as the PRO_CPU (Protocol CPU)
2. 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 at the same priority." This is a limitation of using a ready list designed for a single CPU, on dual-core platforms. [3]

The problem 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 procedure makes the round-robin scheduling less than perfect.

The bottom line for the developer is that the round-robin scheduling is not completely fair for the dual-core ESP32. For many projects, this will not be a noticeable problem. But if it is, there are ways to control the scheduling of your tasks.

Priority Demonstration

Listing 7-3 illustrates a demonstration[4] that makes use of the Lolin 32 ESP device with its built-in SSD1306 OLED display. The program displays three inch-worms that wiggle to and fro, across the display. The worms are driven by CPU consuming tasks, which operate at their configured priorities.

If you are using an ESP32 setup with the SSD1302 OLED using a different configuration, change the following lines. These determine the OLED I2C address, SDA, and SCL gpio pins.

```
0020:     int addr=0x3C,
0021:     int sda=5,
0022:     int scl=4);
```

Experiment 1

In this first experiment, lines 4 through 10 are configured as follows (as downloaded):

```
0004: // Worm task priorities
0005: #define WORM1_TASK_PRIORITY 9
0006: #define WORM2_TASK_PRIORITY 8
0007: #define WORM3_TASK_PRIORITY 7
0008:
0009: // loop() must have highest priority
0010: #define MAIN_TASK_PRIORITY  10
```

Here, the (main) display task runs at priority 10. This priority was chosen because when a worm must be displayed (line 198), it must preempt all other Ready tasks to get the job done. The worm tasks in this experiment run at priorities 9, 8, and 7. Once the worm is drawn on the OLED, the main task becomes blocked again, waiting for the message queue (line 197). This allows tasks lower than priority 10 to resume.

The purpose of this experiment is to demonstrate that while the task at priority 9 executes, the tasks at priority 8 and lower don't. The worm tasks never block, and thus are always in the Ready state.

When the demonstration runs, you will see a display like in Figure 7-2. The top worm (at priority 9) wiggles its way across the top of the display, while the remaining lower priority worms remain at their starting point.

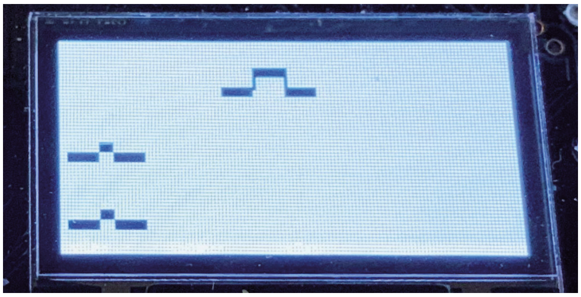


Figure 7-2. The first worm (at task priority 9) marches across the display, while the remaining two remain at the starting point.

All worms are driven by the task named worm_task() in lines 139 to 148. The provided function argument determines whether they are driving worm1, worm2, or worm3, which are C++ InchWorm objects. Each of these objects tracks progress and state. The setup() function creates each of these worm tasks with priorities according to the following macros:

```
0005: #define WORM1_TASK_PRIORITY 9
0006: #define WORM2_TASK_PRIORITY 8
0007: #define WORM3_TASK_PRIORITY 7
```

From the code below, note how the worm_task() never blocks its execution:

```
0139: void worm_task(void *arg) {
0140:   InchWorm *worm = (InchWorm*)arg;
0141:
0142:   for (;;) {
0143:     for ( int x=0; x<800000; ++x )
0144:       __asm__ __volatile__("nop");
0145:     xQueueSendToBack(qh,&worm,0);
0146:     // vTaskDelay(10);
0147:   }
0148: }
```

The task is designed to consume CPU time by running a useless for loop (the __volatile__ keyword is used to avoid the loop from being optimized away by the compiler). At the end of the loop, an attempt is made to queue the address of the worm to be displayed (line 145). This is done without blocking because the time to wait argument is provided with zero. The queued entry triggers the loopTask (main display task) to display the changed worm. If by some chance the queuing of the message fails, it will be picked up on a later loop iteration.

Given that the first (top) worm runs at priority 9, the middle worm at 8, and the bottom worm at priority 7, the OLED display shows the result of the priority 9 task taking all available CPU cycles. The display task runs at priority 10, but blocks shortly after it has performed the display write function, allowing priority 9 and lower levels to schedule again.

The conclusion from this experiment? Priority matters. The priority 9 task never stops executing, resulting in priority 8 and lower tasks waiting to be run. Therefore their corresponding inch-worms don't move.

Experiment 2
For the second experiment, modify the configuration to give the three worms the same priority below 10 but above zero. Leave the main display task at priority 10. I'll use priority 9 here:

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

When you compile and reflash the ESP32, what did you observe? Figure 7-3 is a photo of the display with the three worms marching across the screen, nearly in lock step. If left running long enough, the worms will show some minor differences in progress, however.

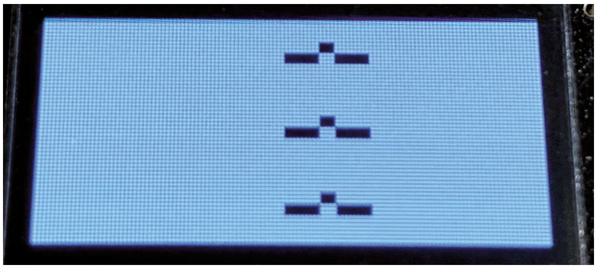


Figure 7-3. The worm tasks all racing at task priority 9.

What can we conclude from this? That tasks running at the same priority get almost equal access to the CPU. Tasks at the same priority get scheduled in a round-robin fashion, as the system tick interrupt occurs. So the tasks tend to get almost equal execution times.

Experiment 3
In this experiment, the three worms remain at the same priority 9 (as in the last demonstration) but the main display task is modified to use priority 9 also:

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

After compiling, reflashing, and running the code, what did you observe?

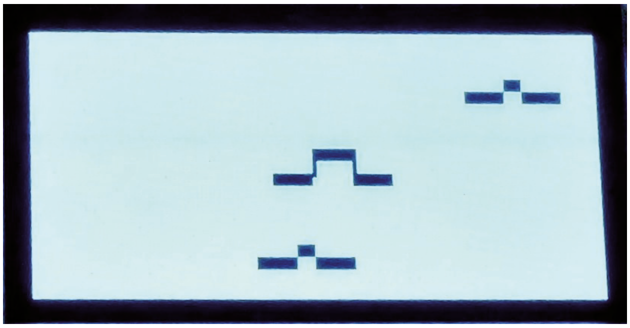


Figure 7-4. The worms marching back and forth when all tasks and the display task run at priority 9.

The bottom worm tends to get the most CPU and is going left unlike the others in the figure. The top worm moves the slowest. The Espressif noted limitation of round-robin unfairness is partly to blame but the other reason is due to the unequal time slices. If the display task were to start a given time slice at the tick interrupt, it might use up 20% before yielding (blocking). Since the system tick doesn't occur until the end of the 1 ms tick cycle, the remaining 80% of that tick is given to the task that is next in line for round-robin.

In this experiment, we see that the scheduling can be unbalanced. Both CPUs are responding to timer and other interrupts. Scheduling changes occur at tick interrupts or when a task blocks (or yields) during its cycle, resulting in unequal time slices.

Experiment 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 worm's task 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(10) 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, reflash, and run this example. What did you observe?

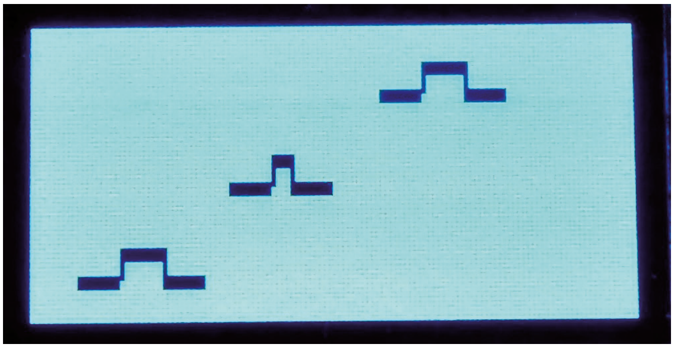


Figure 7-5. Worm wiggle with vTaskDelay(10) added to the worm_task.

The top worm moves the fastest while the bottom worm moves the slowest. The top worm with priority 9 gets the 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 preempted whenever the higher priority 9 task becomes ready again. This is why the top worm moves fastest. The middle worm can sometimes preempt the priority 7 task, so it tends to be faster than the bottom worm.

```
// worms1.ino
// MIT License (see file LICENSE)

// 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

#include "SSD1306.h"

class Display : public SSD1306 {
  int w, h; // Width, height
```

```
public:
  Display(
    int width=128,
    int height=64,
    int addr=0x3C,
    int sda=5,
    int scl=4);
  int width() { return w; }
  int height() { return h; }
  void lock();
  void unlock();
  void clear();
  void init();
};

class InchWorm {
  static const int segw = 9, segsw = 4, segh = 3;
  Display& disp;
  int worm;
  int x, y; // Coordinates of worm (left)
  int wormw=30;
  int wormh=10;
  int dir=1;  // Direction
  int state=0;
public:
  InchWorm(Display& disp,int worm);
  void draw();
};

InchWorm::InchWorm(Display& disp,int worm)
: disp(disp), worm(worm) {
}

void
InchWorm::draw() {
  int py = 7 + (worm-1) * 20;
  int px = 2 + x;

  py += wormh - 3;

  disp.setColor(WHITE);
  disp.fillRect(px,py-2*segh,3*segw,3*segh);
  disp.setColor(BLACK);

  switch ( state ) {
    case 0: // _-_
```

```
   disp.fillRect(px,py,segw,segh);
   disp.fillRect(px+segw,py-segh,segsw,segh);
   disp.fillRect(px+segw+segsw,py,segw,segh);
   break;
  case 1: // _^_ (high hump)
   disp.fillRect(px,py,segw,segh);
   disp.fillRect(px+segw,py-2*segh,segsw,segh);
   disp.fillRect(px+segw+segsw,py,segw,segh);
   disp.drawLine(px+segw,py,px+segw,py-2*segh);
   disp.drawLine(px+segw+segsw,py,px+segw+segsw,py-2*segh);
   break;
  case 2: // _^^_ (high hump, stretched)
   if ( dir < 0 )
    px -= segsw;
   disp.fillRect(px,py,segw,segh);
   disp.fillRect(px+segw,py-2*segh,segw,segh);
   disp.fillRect(px+2*segw,py,segw,segh);
   disp.drawLine(px+segw,py,px+segw,py-2*segh);
   disp.drawLine(px+2*segw,py,px+2*segw,py-2*segh);
   break;
  case 3: // _-_ (moved)
   if ( dir < 0 )
    px -= segsw;
   else
    px += segsw;
   disp.fillRect(px,py,segw,segh);
   disp.fillRect(px+segw,py-segh,segsw,segh);
   disp.fillRect(px+segw+segsw,py,segw,segh);
   break;
  default:
   ;
 }
 state = (state+1) % 4;
 if ( !state ) {
  x += dir*segsw;
  if ( dir > 0 ) {
   if ( x + 3*segw+segsw >= disp.width() )
    dir = -1;
  } else if ( x <= 2 )
   dir = +1;
 }
 disp.display();
}

Display::Display(
 int width,
 int height,
 int addr,
 int sda,
 int scl)
: w(width), h(height),
 SSD1306(addr,sda,scl) {
}

void
Display::init() {
 SSD1306::init();
 clear();
 flipScreenVertically();
 display();
}

void
Display::clear() {
 SSD1306::clear();
 setColor(WHITE);
 fillRect(0,0,w,h);
 setColor(BLACK);
}

static Display oled;
static InchWorm worm1(oled,1);
static InchWorm worm2(oled,2);
static InchWorm worm3(oled,3);
static QueueHandle_t qh = 0;
static int app_cpu = 0; // Updated by setup()

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);
 }
}

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);

 delay(1000);  // Allow USB to connect
 printf("worms1.ino: CPU %d\n",app_cpu);
}

void loop() {
 InchWorm *worm = nullptr;

 if ( xQueueReceive(qh,&worm,1) == pdPASS )
  worm->draw();
 else
```

```
    delay(1);
}
```

Listing 7-3. The worms1.ino demonstration program.

Priority Configuration

While we have not yet covered interrupt use within the ESP32, be aware that the header file FreeRTOSConfig.h configures priorities for the platform. The header defines the following priority macro values. The Arduino compiled in values are shown:

1. configMAX_PRIORITIES = 25
2. configKERNEL_INTERRUPT_PRIORITY = 1
3. configMAX_SYSCALL_INTERRUPT_PRIORITY = 3

The first macro defines the maximum number of priorities available. This declares that valid priority numbers range from 0 to 24 inclusive.

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(), as we will see when interrupts are covered. With the configured values shown, those FreeRTOS functions can only be called from interrupt priorities 1 to 3 inclusive. When no FromISR() calls are made, the ISR may freely operate at priorities 4 through 24 inclusive.

Scheduler Review

Let's review the FreeRTOS scheduling algorithm as configured by Arduino in the simplest possible terms:

1. The FreeRTOS always searches from highest priority (24) to lowest priority (0) for a task to execute.
2. To be eligible to execute, the task must be in the Ready state (in other words, the task is not blocked or suspended).
3. The scheduler executes the selected ready task. If there are no ready tasks, the idle task is run.
4. At the next system tick interrupt (or block/yield), the scheduler must again choose the next task to run, from highest to lowest priority sequence.
5. When there are multiple tasks ready to run at a given priority, they are selected in round-robin fashion. If there are no other tasks at this priority, the same task is re-selected to run.

Apart from the ESP32 wrinkle where the round-robin is not perfectly sequenced due to the dual-core Espressif customization, this is how scheduling occurs within Arduino.

Summary

What can we conclude from these experiments? Real-time priority is not so simple after all. The consequence is that if 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 for others. The problem depends upon the nature of your application.

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 perform blocking calls. 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.

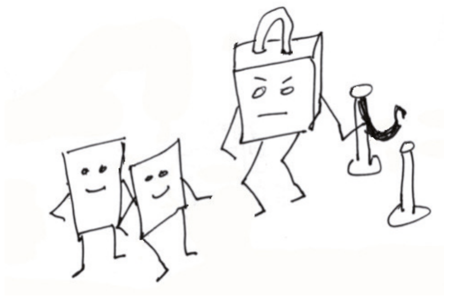
Exercises

1. What is the most urgent and least urgent priority for the ESP32 Arduino?
2. Do you need to call vTaskStartScheduler() for Arduino?
3. When does the FreeRTOS scheduler get invoked in the course of a program's execution?
4. What kind of scheduling occurs among tasks of equal priority?
5. How do you create a ready-to-go task and unleash it only when it is time for it to start?
6. What causes the execution of a task to be pre-empted?
7. On the ESP32, what is the maximum time slice in ms?
8. When does less than a full-time slice occur?
9. Is the call to taskYIELD() a blocking call? Why or why not?
10. Is the call to the Arduino delay() (or FreeRTOS vTaskDelay()) a blocking call? Why or why not?
11. How do you invoke the FreeRTOS scheduler directly?
12. How does a task become CPU starved?
13. Does a call to taskYIELD() ever result in a higher priority task running? Why or why not?

Web Resources

[1] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/taskcreate/taskcreate.ino
[2] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/taskcreate2/taskcreate2.ino
[3] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/freertos-smp.html
[4] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/worms1/worms1.ino

Chapter 8 • Mutexes



One task at a time, please!

The term mutex is derived from the words mutual and exclusion. Given that FreeRTOS already supports binary semaphores, you might wonder why another semaphore-like facility is needed. What is the distinguishing talent of a mutex that binary semaphores lack? This chapter will highlight that talent and describe how it helps to solve a difficult problem.

Exclusion Principle

Like the binary semaphore, the mutex is a mechanism, which provides for mutual exclusion. Like binary semaphores, the exclusion principle is based upon agreement only. The mutex itself does not exclude or grant access. It is assumed that by agreement, a task will not access the protected resource unless it has gained success from the mutex. In FreeRTOS terminology, the task has "taken" the mutex. Once the mutex has been "taken", the caller becomes the sole owner of that resource.

When the owner is finished with the resource, it releases access to it by giving the mutex. By giving the mutex, another task will be able to take it. Using this protocol, no more than one task will have simultaneous access to the protected resource.

A use case might be two tasks driving I2C devices on the same bus. The bus is common to the devices, yet the bus can only carry one transaction at a time. A mutex allows two independent tasks to communicate with their slave devices, yet prevent collisions on the shared bus.

What's the Problem?

So far, you haven't read anything new – this has all been review. The question to be answered now, is what is the problem to be solved by the mutex that the binary semaphore cannot? There is a reason that this discussion follows the chapter about priorities. The problem is priority based.

Imagine three tasks A, B, and C. Tasks A and B both need to use a shared I2C bus protected by a binary semaphore for mutual exclusion. Task A runs at task priority 1, while task B operates at 3 and another independent task C at priority 2. Bear with me on the details, they are important:

1. Task B (priority 3) happens to be blocked waiting for a message queue.
2. Task C (priority 2) is also currently blocked for some reason (the reason is unimportant).
3. Task A (priority 1) executes because no higher priority tasks are ready. It locks the semaphore for the I2C bus and becomes the owner.
4. Task C (priority 2) becomes unblocked and pre-empts Task A due to priority.
5. Task B (priority 3) next happens to receive a message from the queue and becomes unblocked (pre-empting Task C).
6. Task B (priority 3) blocks trying to lock the I2C bus semaphore, that Task A owns. Task C at priority 2 now resumes.
7. Task B (priority 3) remains blocked waiting for the lock on the I2C bus that Task A holds.

The problem is that Task C is running at priority 2. If priority in the system was truly respected, Task B (priority 3) *should be running*. But Task B cannot run because of the lock owned by a lower priority 1 Task A. Figure 8-1 illustrates the situation. Because of the lock held by Task A, Task C is running at the end instead of the higher priority Task B.

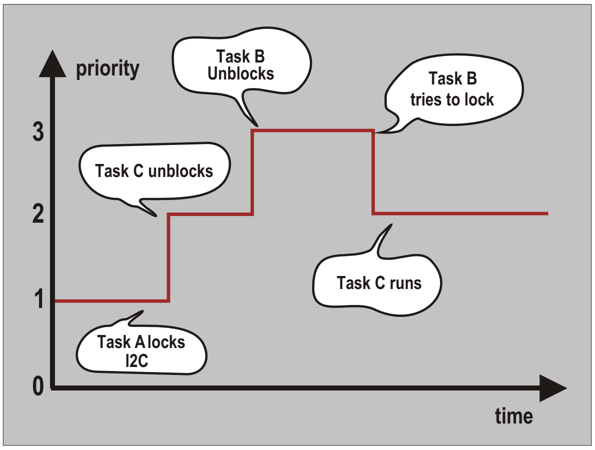


Figure 8-1. Because Task A owns the semaphore, Task C ends up running instead of the higher priority Task B (priority inversion).

The Mutex Solution
The solution to the problem presented is to have Task A run at priority 3. Yet the designer has weighed the needs of the application and decided Task A must run at priority 1.

The special talent of the mutex is to temporarily boost the lower priority task owning the lock until the lock is released. If a mutex were used instead of the binary semaphore, the sequence of the events would change to the following:

1. Task B (priority 3) happens to be blocked waiting for a message queue.
2. Task C (priority 2) is also currently blocked for some reason (the reason is unimportant).

3. Task A (priority 1) executes because no higher priority tasks are ready. It locks the mutex for the I2C bus and becomes the owner.
4. Task C (priority 2) becomes unblocked and pre-empts Task A due to priority.
5. Task B (priority 3) happens to receive a message from the queue and becomes unblocked (pre-empting Task C).
6. Task B (priority 3) tries to lock the I2C bus, which Task A owns. The special talent of the mutex is to temporarily boost the priority of the lock owner (Task A) to priority 3.
7. Task A (now at priority 3) can run to the point of releasing the mutex. Once the mutex is released, it then returns to priority 1.
8. Task B (priority 3) now owns the lock and executes (while Task C remains ready but not executing).

In the mutex example, the mutex boosts Task A to priority 3 temporarily because Task B was at priority 3 and wanting that mutex. This temporary boost permitted Task A to run until it released the mutex. Once the mutex is released, Task A returns to priority 1 and then Task B pre-empts it. Task B then executes while owning the lock.

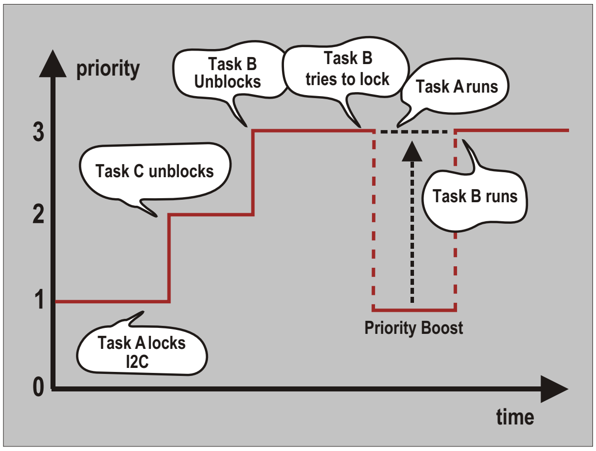


Figure 8-2. Task A gets a priority boost when Task B tries to lock, resulting in Task A unlocking allowing the higher priority Task B to run.

Priority Inversion

What was the essential difference between the binary semaphore and the priority talented mutex? In the binary semaphore example, the priority 2 task (C) ends up executing while the higher priority 3 Task B languishes, waiting for the lower priority Task A. In other words, a lower priority Task C was caused to run ahead of a higher priority task. This is highly undesirable and known as priority inversion.

When the mutex was used, the highest priority Task B ends up executing instead of Task C. This is how priority-based scheduling is supposed to operate.

What if Task A never released the mutex? Or runs for a long time without releasing it? This is indeed a problem because Task A will run with the boosted priority if another process tries to take the same mutex. A mutex avoids priority inversion by a temporary priority boost but this by itself is no guarantee that the system design is problem-free. That responsibility still rests with the program designer.

Creating a Mutex
Creating a mutex is much like creating a binary semaphore:

```
        SemaphoreHandle_t xSemaphoreCreateMutex(void);
```

The call simply returns a handle to the mutex created. For a non-Arduino environment, you can also statically allocate and create a mutex:

```
        static StaticSemaphore_t m1;
        ...
        SemaphoreHandle_t h1 =  xSemaphoreCreateMutexStatic(&m1);
```

Either way, you obtain a handle to a binary semaphore with the special priority boost talent of the mutex.

Note: The *mutex* is created in the *given* (unlocked) state (unlike the binary semaphore, which is created in the *taken* state). To lock a mutex, you must "take" it. When you are finished with the protected resource and want to unlock it, you must "give" it.

Give and Take
Apart from the initial state of a mutex, it is used in the same way as a binary semaphore:

```
        BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

The give operation returns either pdPASS or pdFAIL. The give operation fails if the semaphore has already been given.

The take operation requires the xTicksToWait parameter to determine the action to take when the mutex is found locked by another task.

```
        BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore,TickType_t
        xTicksToWait);
```

Specify the number of ticks to wait, the macro portMAX_DELAY or zero for xTicksToWait. The value pdPASS is returned when successful, otherwise pdFAIL. A task should only call xSemaphoreTake() on a mutex that it has not already taken.

Deleting a Mutex
A mutex can be deleted (including a statically allocated one) with the following call:

```
        void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);
```

Despite the function name, this function can be used to delete any of:

- Binary semaphore
- Counting Semaphore
- Mutex
- recursive Mutex (to be discussed)

Note: A handle to a semaphore or mutex, should never be deleted if there are tasks blocked on it. Doing so may result in a fatal error.

Demonstration
A practical demonstration using a mutex to protect a common I2C bus is provided in Listing 8-1.[1] This demonstration makes use of a pair of PCF8574 or PCF8574A GPIO extender chips, to provide (you guessed it) extra GPIO pins. These chips are wonderful because of their low cost and are so easy to use. They also operate at +3.3 volts at a maximum of 100 µA. Best of all, these are available in PDIP form (Plastic Dual Inline Package).

The program demonstrates two tasks driving an LED via each expander chip. Because the tasks are fully independent of each other and using a common I2C bus, the mutex is used to protect the bus from simultaneous access.

PCF8574 Chip
The PCF8574 chip is a great way to add 8 more GPIO pins to any project – input, output, or a mix of both. Because the chip is accessed through the I2C bus, it naturally has timing limitations based upon I2C bus transactions. So when planning a project, allocate the slowest I/O pins to the GPIO extender. Figure 8-3 illustrates the PDIP pinout.

There is almost no software configuration required to use this chip. Writing requires no configuration – just write the 8 bits of data out to the device. For inputs, write 1-bits to the input pins and either state to the remaining outputs. The 1-bit outputs activate an internal pull-up resistor/regulator circuit within the chip. The input levels are then decided by the attached circuit to pull the voltage low or to leave it high. It takes no more than 300 µA to pull the pin down to ground potential. All that is left to do is to read the port.

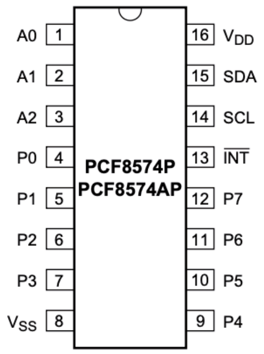


Figure 8-3. Pinout of the PCF8574/PCF8574A chip.

Pins A0, A1, and A2 decide the I2C address for the chip. The presented project will ground all three pins for the first chip and ground A1 and A2 for the second with A0 tied high. Depending upon which chip you use, this configures the chips to have I2C addresses:

PCF8574 – 0x20 and 0x21
PCF8574A – 0x38 and 0x39

Pins for SDA and SCL are the familiar I2C bus pins. There is an INT pin for signalling the MPU for interrupts, which is not used this time. The INT pin can be useful for GPIO inputs. The power connections are +3.3 volts for V_{DD} and ground for V_{SS}.

LED Drive

One quirk of the PCF8574 chip is that it cannot *source* current from a GPIO pin beyond 300 µA maximum (nxp.com). To drive an LED you must *sink* the current instead (Figure 8-4). The maximum sink current is 25 mA. This is more than enough to drive an LED. This, of course, affects the software, since a 0-bit must be written to the GPIO pin to light the LED (in active low configuration).

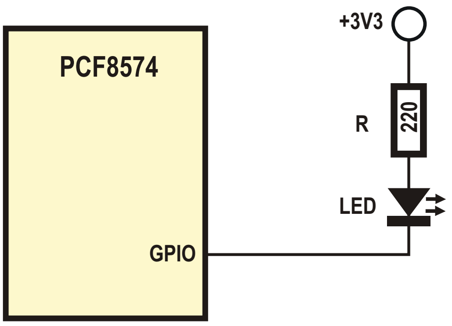


Figure 8-4. PCF8574 must sink the current for the LED, in active low configuration.

The project schematic is shown in Figure 8-5. Essentially just add a pair of PCF8574 (or PCF8574A) chips to the I2C bus, and power them from the ESP32. Be sure to double-check the chip orientation when wiring them up.

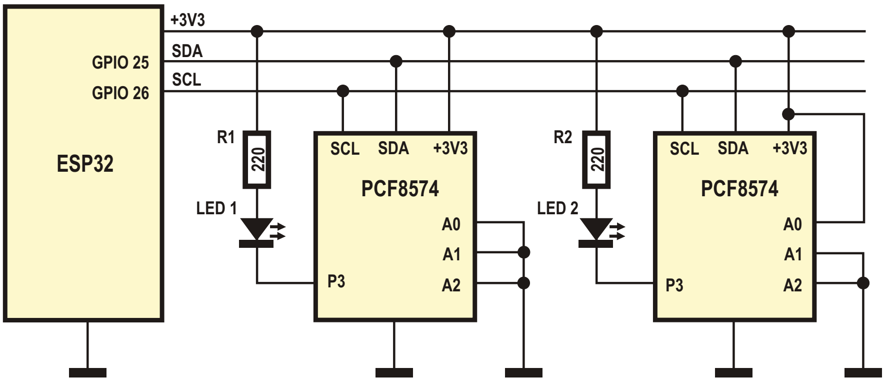


Figure 8-5. Schematic of the demonstration using two PCF8574 GPIO extender chips.

Code Break Down

The downloaded code is preconfigured to use GPIO 25 and 26 as the I2C SDA and SCL respectively. Change lines 4 and 5 if necessary.

The PCF8574 chip comes in two varieties and either can be used when both are the same. If you mix the types, you must modify the source code slightly to make it work. The three address pins of the PCF8574 configure the I2C address. In this project, these are all wired to ground, except for the second chip (Figure 8-5). The second chip has A0 wired to +3.3V so that A0 registers as a 1-bit.

Depending upon the chip used, with the address pins A0 through A2 grounded, the addresses are:

PCF8574 – 0x20
PCF8574A – 0x38

The second chip is wired with A0 wired to +3.3V so that the addresses are:

PCF8574 – 0x21
PCF8574A – 0x39

The setup() function performs some routine initialization and the creation of a mutex:

```
0107:   mutex = xSemaphoreCreateMutex();
```

The mutex is created initially in the *unlocked* (given) state, unlike the binary semaphore. A binary semaphore and a mutex differ in this manner. Let's now turn our attention to the task function led_task().

Line 51 obtains the I2C address of the chip that will be driven. This value comes from line 120 or 131, depending upon the task. Before the led_task() begins its loop, it tests for the presence of the I2C expander chip in lines 55 to 72.

Lines 55 and 72, invoke the packaged up mutex take, and give functions respectively. After gaining exclusive use of the I2C bus by returning from lock_i2c() in line 55, the I2C transaction is begun in line 58. The program requests a read of 1 byte from our extender chip (line 59) and tests the result in lines 50 and 61. If the number of bytes returned is greater than zero, we know that the chip responded to our read request. The read in line 62 just pulls out the byte and discards it.

If the chip fails to respond to the read request, the value returned for line 60 will be zero. This causes the program to report the problem to the Serial Monitor in lines 69 and 70. The current task is also deleted in line 76 when this happens since there is no point in going further.

Troubleshooting
If you see a message on the Serial Monitor of the form:

```
I2C address 0x21 not responding.
```

Then you need to find out why. The suggested troubleshooting sequence is:

- Check the orientation of the chips (power off immediately if incorrect!)
- Check +3.3V power and ground on the PCF8574 chips.
- Check that SDA is wired to GPIO 25 (or custom I2C_SDA pin)
- Check that SCL is wired to GPIO 26 (or custom I2C_SCL pin)
- That SDA and SCL are wired correctly to both chips.
- Check that A0, A1, and A2 are grounded for the first chip.
- Check that A1 is +3.3V and that A1 and A2 are grounded for the second chip.
- Check for faulty connections or wiring.
- If you used custom GPIO values for the I2C_SDA and I2C_SCL, then check that these are usable GPIOs and not in conflict with other built-in components.

Blink Loop
Once the led_task() gets past the initial I2C checkout (lines 55 to 77), it enters the "blink loop" in lines 82 to 95. This loop is fairly straight forward performing the following:

1. Lock the I2C bus (line 83)
2. Start an I2C write to the PCF8574 peripheral (line 88)
3. Write a low to bit 3 (GPIO P3 on the chip) if the LED is to be on, or otherwise a 1-bit. All other bits are always written as 1-bits here.
4. The I2C transaction is ended (line 90).
5. And the I2C bus is unlocked (line 91).
6. The delay() used (line 94) is different for the two tasks. This makes it obvious that both of the GPIO expanders are being driven independently.

Running the Demonstration

The first time you run the demonstration, start the Serial Monitor, and look for error messages after the flash and go. If all went well, you should see:

```
mutex.ino:
Testing I2C address 0x20
I2C address 0x20 present.
Testing I2C address 0x21
I2C address 0x21 present.
LED 0x20 on
LED 0x21 on
LED 0x21 off
LED 0x20 off
LED 0x21 on
LED 0x20 on
LED 0x21 off
...
```

After a short time, the LEDs should be blinking independently in an uncoordinated fashion. If the LEDs are not lighting, then make sure that the polarity of the LED is correct. No harm will result if you just turn them around.

Figure 8-6 is a photo of a breadboard setup. This uses the ESP32 Wemos Lolin device, though there is no requirement to use that dev board. The OLED is not used.

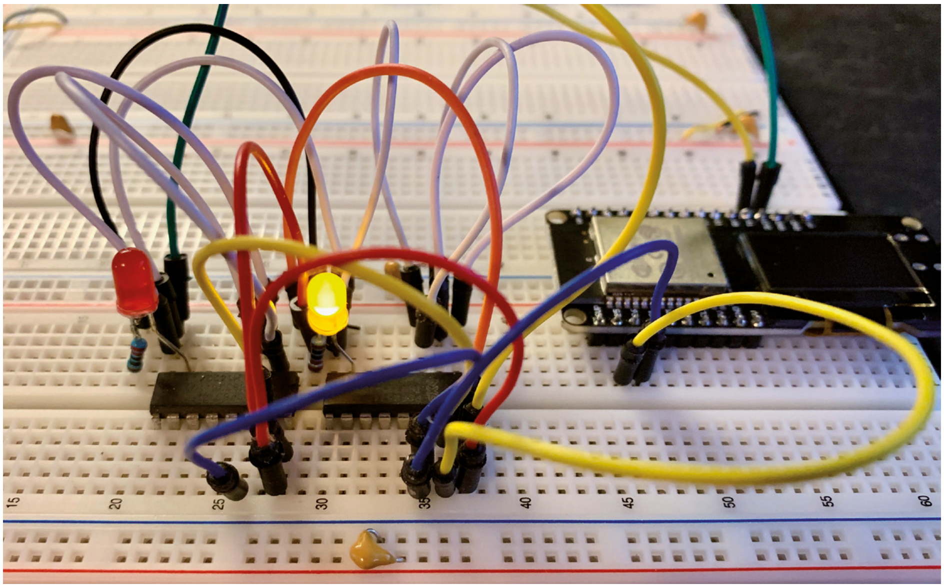


Figure 8-6. The demonstration wired up on a breadboard with LED + resistor pairs wired into the PCF8574 chips at pin P3.

```
// mutex.ino

// GPIOs used for I2C
#define I2C_SDA       25
#define I2C_SCL       26

// Set to 1 for PCF8574A
#define PCF8574A      0

#include <Wire.h>

#if PCF8574A
// Newer PCF8574A addresses
#define DEV0          0x38
#define DEV1          0x39
#else
// Original PCF8574 addresses
#define DEV0          0x20
#define DEV1          0x21
#endif

static int app_cpu = 0;
static SemaphoreHandle_t mutex;
static int pcf8574_1 = DEV0;
static int pcf8574_2 = DEV1;

//
// Lock I2C Bus with mutex
//
static void lock_i2c() {
  BaseType_t rc;

  rc = xSemaphoreTake(mutex,portMAX_DELAY);
  assert(rc == pdPASS);
}

//
// Unlock I2C Bus with mutex
//
static void unlock_i2c() {
  BaseType_t rc;

  rc = xSemaphoreGive(mutex);
  assert(rc == pdPASS);
}

```

```
//
// I2C extender blink task:
//
static void led_task(void *argp) {
  int i2c_addr = *(unsigned*)argp;
  bool led_status = false;
  int rc;

  lock_i2c();
  printf("Testing I2C address 0x%02X\n",
    i2c_addr);
  Wire.begin();
  Wire.requestFrom(i2c_addr,1);
  rc = Wire.available();
  if ( rc > 0 ) {
    Wire.read();
    Wire.beginTransmission(i2c_addr);
    Wire.write(0xFF); // All GPIOs high
    Wire.endTransmission();
    printf("I2C address 0x%02X present.\n",
      i2c_addr);
  } else  {
    printf("I2C address 0x%02X not responding.\n",
      i2c_addr);
  }
  unlock_i2c();

  if ( rc <= 0 ) {
    // Cancel task if I2C fail
    vTaskDelete(nullptr);
  }

  //
  // Blink loop
  //
  for (;;) {
    lock_i2c();
    led_status ^= true;
    printf("LED 0x%02X %s\n",
      i2c_addr,
      led_status ? "on" : "off");
    Wire.beginTransmission(i2c_addr);
    Wire.write(led_status ? 0b11110111 : 0b11111111);
    Wire.endTransmission();
    unlock_i2c();

    // Use different delays per task
    delay(i2c_addr & 1 ? 500 : 600);
  }
}

//
// Initialize
//
void setup() {
  BaseType_t rc;  // Return code

  app_cpu = xPortGetCoreID();

  // Create mutex
  mutex = xSemaphoreCreateMutex();
  assert(mutex);

  // Start I2C Bus Support:
  Wire.begin(I2C_SDA,I2C_SCL);

  delay(2000);  // Allow USB to connect
  printf("\nmutex.ino:\n");

  rc = xTaskCreatePinnedToCore(
    led_task,   // Function
    "led_task1",// Name
    2000,       // Stack size
    &pcf8574_1, // Argument
    1,          // Priority
    nullptr,    // Handle ptr
    app_cpu     // CPU
  );
  assert(rc == pdPASS);

  rc = xTaskCreatePinnedToCore(
    led_task,   // Function
    "led_task2",// Name
    2000,       // Stack size
    &pcf8574_2, // Argument
    1,          // Priority
    nullptr,    // Handle ptr
    app_cpu     // CPU
  );
  assert(rc == pdPASS);
}

```

```
// Not used:
void loop() {
  vTaskDelete(nullptr);
}
```

Listing 8-1. Mutex demonstration using GPIO extenders PCF8574 or PCF8574A

Recursive Mutexes
Some might wonder at this point what other possible facility is needed for a mutex? The answer to that lies in how the problem manifests itself in packaged software.

In the demonstration, the two independent tasks performed a simple lock and unlock to prevent I2C bus conflict. The operations involved were simple. But what happens if you have several devices and several layers of software? As the complexity increases, a scenario often develops that the subroutine must know if the mutex has already been locked or not, and if not, lock it. Upon return, the reverse must happen.

In large applications, managing these seemingly trivial problems starts to add to your woes. The subroutine writer just wants to make sure that the resource is locked. If it is already locked, then skip the locking and if not, do so now. Upon subroutine return, this must be undone. The task will block forever if the mutex was already locked and it is an error to unlock an already unlocked mutex. How can this annoying issue be resolved?

The recursive mutex solves this problem by keeping track of a nested lock count. When locking, if the count goes from 0 to 1, then a mutex lock event must occur. If the mutex is already locked by this task, simply increment the internal count from 1 to 2, etc. When the subroutine is returning, it can recursively unlock the mutex. If the mutex was nested locked (by the calling task), then the count can be decremented from 2 to 1, for example. At the outer layer, another unlock when the count is 1 (by the same task), means that an actual mutex unlock (give) must occur.

The only responsibility that a user of a recursive mutex has is to always unlock as many times as it has been locked. If this gets bungled, then one of two things will happen:

- The code will have an overly locked mutex, which thus stays locked (not enough unlock (give) operations have been performed).
- The code will attempt to unlock too many times (returning an unexpected error).

All of this might sound rather academic. For simple applications, a few screens worth of code easily identifies all the points where locks and unlocks occur. On large projects, however, especially with multiple contributing programmers, the situation is vastly different.

Recursive Mutex API
To create and use a recursive mutex, you simply add the name component "Recursive" as shown below to the already familiar API calls:

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void);
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(StaticSemaphore_t
pxMutexBuffer)
BaseType_t xSemaphoreGiveRecursive(SemaphoreHandle_t xMutex);
BaseType_t xSemaphoreTakeRecursive(
  SemaphoreHandle_t xMutex,
  TickType_t xTicksToWait
);
```

The operation of the mutex beyond being recursive is otherwise identical to a plain mutex.

Note: A recursive mutex must *only* be taken with xSemaphoreTakeRecursive(), and never be used with the xSemaphoreTake(). Likewise, a recursive mutex must only be given with xSemaphoreGiveRecursive() and not with xSemaphoreGive().

Deadlock Avoidance and Prevention
When a task must obtain multiple locks, there is potential for deadlock. These can be quite time consuming to debug and may not always show up during testing. Consequently, it is best to put some effort into prevention upfront.

To allow for the best utilization of shared resources, avoid locking until the last possible moment and obtain your locks all at once. Otherwise, a locked resource could be preventing other tasks from proceeding when they otherwise might.

Always lock multiple locks in the same sequence to avoid circular dependencies. Never have task 1 lock A first and task 2 lock B first and then each task try for the other. If both tasks try for lock A first, then only one task will try for lock B.

When several locks are involved, that alone will not be enough because the group of resources involved is not always the same. There are several algorithms available – the Bankers Algorithm is a popular choice. In systems with deadlock detection, another approach is to unlock all resources when a deadlock occurs and then retry.

Recursive Mutex Usage
Some practitioners feel that using a recursive mutex is bad practice.[2] Like most things, a recursive mutex can be abused. It is probably best that new designs avoid using the recursive mutex initially. Upgrade to the recursive form only when it becomes necessary to solve a library nesting issue.

It should be understood however that in a complex system, the recursive mutex can paint you into a corner. Imagine that your recursive mutex is now two or three levels deep and your subroutine must now lock yet another mutex. To perform deadlock avoidance, that subroutine may need to unlock the recursive mutex to re-attempt locking. The recursive unlock will fail if the lock is already nested.

At the end of the day, the system designer must consider carefully if the risks are worth the benefit of recursion. Another factor to be considered is how much will the system's code be changed by others that were not involved in the original design? Will they make unsafe assumptions? Do them a favour and leave adequate documentation in the comments.

Summary

A mutex is much more than the simple binary semaphore. FreeRTOS was designed to keep the highest priority tasks running at every turn. When a high priority task needs a lock that is held by a lower priority task the best solution is to allow that low priority task to be temporarily boosted so that the lock will be released. This keeps the high priority tasks running as intended.

The best answer is simply to design the system in a way that a low priority task never holds an important lock. Yet this may prove to be difficult. A low priority task may need to read an I2C temperature sensor every few seconds, for example. At some point, it will have to lock the I2C bus and perform the read. If this happens during high priority I2C operation, then the temporary priority boost comes to the rescue.

A recursive mutex solves the additional problem of nested locking. Some feel that this introduces "code smell". Others believe that it is an acceptable way to solve the lock nesting problem that often occurs in libraries and some usage patterns. Whatever you decide, give it due consideration.

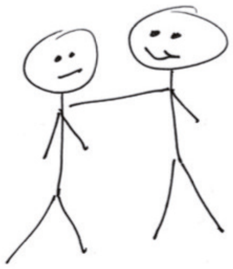
Exercises

1. How does a low priority task interfere with a high priority task when it holds a shared binary semaphore? What is the impact of this?
2. If the non-recursive mutex has already been locked by a task, is it an error for the same task to lock it again?
3. How does the mutex prevent priority inversion?
4. After the temporary priority boost, at what point does the task return to its original priority?
5. What happens if a recursive mutex is not unlocked (given) the same number of times as it was locked?
6. When is it unsafe to delete a recursive mutex?
7. Is the initial state of a mutex different from a binary semaphore? If so, how?
8. Why must the PCF8574 drive a LED by sinking current instead of sourcing it?

Web Resources

[1] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/mutex/mutex.ino
[2] https://github.com/isocpp/CppCoreGuidelines/issues/871

Chapter 9 • Interrupts



Son, never interrupt your wife while she's telling a joke.
Else you'll never hear the end of it.

A telephone call is an example of an interrupt that occurs in everyday life. If you choose to answer the call, you stop what you're doing and answer the phone. A program accepts interrupts in the same manner – it stops executing the current line of code and calls upon an ISR (Interrupt Service Routine) to service it. When the interrupt has been serviced, the CPU resumes where it left off.

Interrupts are not necessarily new to the Arduino enthusiast. But for the benefit of the new student, this chapter will examine interrupt related concepts. Then we'll examine how FreeRTOS works within that framework.

Characteristics of an ISR

What makes the ISR so special? Some qualities of an ISR include:

- The ISR can be invoked at almost any time, making it an *asynchronous* routine.
- The ISR is called using an allocated ISR stack (for ESP32).
- Because of its asynchronous nature, the code must not call non-reentrant functions.
- An ISR blocks lower priority ISRs while the current one executes.
- Because of priority blocking its execution must be short.
- It must not block its execution in a FreeRTOS function call.
- An ISR does not execute as part of a task.
- ISRs often have special calling requirements.

The Asynchronous ISR

Aside from special platform functions that inhibit interrupts or the disabling of a particular interrupt, there is no control over when the ISR might be invoked. For example, if a GPIO input is configured for rising edge interrupts and is enabled, then the ISR will be called whenever the CPU senses a rising edge, and when other higher priority ISRs are not pending. When higher priorities prevail, the lower priority ISRs will be serviced after the higher priority ISRs have returned.

When the ISR is invoked by the CPU, the current registers and CPU state are saved before performing the special function call into the ISR. This allows the interrupted code to be resumed after the ISR has returned.

The ISR Stack

The ISR stack convention varies with different hardware and software platforms. For the Arduino ESP32, there is a separately allocated stack for ISR routines to use. The size of that stack is limited to 1536 bytes based upon the definition of the FreeRTOS configuration macro named CONFIG_FREERTOS_ISR_STACKSIZE. This stack is potentially used by nested ISR routine calls (an interrupt can interrupt other ISRs due to priority). The consequence of this is that your ISR may not have the full stack allocation of 1536 bytes available for your own use. Your ISR should use the stack frame sparingly.

Non-Reentrant Routine Calls

If you think about this problem, the conclusion is obvious. But the new student may not see this coming. When your ISR routine is called, perhaps for a GPIO input level change, you have no idea what piece of code was interrupted. The interrupted task may have been in the middle of requesting a storage allocation from malloc(), for example. If an interrupt happens while in the middle of malloc() adjusting an internal linked list of memory blocks, then calling malloc() (recursively) from the ISR can muddle things badly. For this reason, an ISR must only call reentrant routines.

A function is said to be reentrant if it can be invoked multiple times (nested calls) without corrupting the outcome of the prior calls in progress. A simple example of a non-reentrant and reentrant function is the rand() vs rand_r() as provided by the newlib library (on ESP32).[1]

```
    int rand(void);             // non-reentrant

    int rand_r(unsigned *seed);  // reentrant
```

When rand() is used, the result depends upon some internal seed value (state). Thus if rand() were in the middle of computing a random number, when the ISR() was entered, and the ISR called rand() as well, then the nested call will disturb the result for the original call. When the reentrant version rand_r() is called instead, the result depends only upon the supplied argument passed in the call.

ISR Priorities

The ESP32 hardware like many other platforms provides for priority-based interrupts. Some interrupts are simply more critical than others and are thus serviced first. The ESP32 core provides 32 interrupts for each CPU core, with configured priority levels. If a low priority interrupt is started when a higher priority interrupt arrives, then a new stack frame is created as part of the call to the higher priority ISR. When the higher priority ISR returns, the lower priority interrupt processing resumes.

As long as the higher priority ISR executes, the lower priority ISR is suspended. If the high priority ISR takes too long to execute, the lower priority ISR may miss the event it was hoping to capture (perhaps the GPIO level of the input). For this reason, ISR routines should be short.

This interrupt priority is distinct and separate from the FreeRTOS task priority. The ISR executes separately from tasks and as such does not have a task priority.

Short ISR Routines
Aside from the possibility of missing critical events, there is another reason to desire short ISRs. There may not be enough time and stack to keep executing them. This can result in lost interrupts or excessive stack nesting.

As an example, the ESP32 system tick interrupt occurs at 1 ms intervals. This means that the ISR is invoked 1000 times per second. If during the tick processing the ISR itself took another 1 ms to execute, the next tick may be missed altogether, resulting in only 500 ticks per second. This is bad enough but what happens if the interrupts nest? An ISR must clear an interrupt flag as part of its processing. Clearing the flag permits another interrupt to occur. So if before the ISR can return, it clears the flag and another interrupt immediately occurs, the calls nest on the stack. If nesting is allowed to continue, the stack will eventually be overrun.

A short ISR on the other hand should be able to service the current interrupt, clear the flag and return before further interrupts occur.

ISR is not a Task
Because the ISR is not an executing as part of a task, there are implications for FreeRTOS services. For example, there is no current task priority (*interrupt* priority is different). We've already noted that the ISR uses a separately allocated stack. This is helpful because it means that your tasks don't have to share their stack with interrupts being serviced.

Special ISR Code
On most hardware platforms there are special requirements for ISR code. The nature of the interrupt handling often requires a particular register save and call convention. On the ESP32, the ISR is simply declared like this:

```
void IRAM_ATTR my_isr() {
  ...
}
```

The macro IRAM_ATTR indicates to the compiler that the function my_isr() must be placed into IRAM (Instruction Random Access Memory). This ensures that your ISR will be executed immediately when the interrupt occurs. Otherwise, a pending flash erase or write operation might busy the flash memory and prevent your ISR routine from executing for up to hundreds of milliseconds.

The IRAM memory available is limited, which is another reason to keep the ISR short. On the ESP32, there is approximately 128 kB IRAM physically available for application use, but some of that is likely preallocated.

ESP32 Arduino GPIO Interrupts

Declaring an ISR for GPIO interrupts is quite easy in Arduino on the ESP32. The following code illustrates how to capture a rising pulse on GPIO input 14 (GPIO_PULSEIN). The necessary ISR routine and setup functions are shown below:

```
#define GPIO_PULSEIN  14

void IRAM_ATTR isr_pulse() {
   ...
}

void setup() {

  pinMode(GPIO_PULSEIN,INPUT_PULLUP);
  attachInterrupt(GPIO_PULSEIN,isr_pulse,RISING);
```

In the setup routine, simply configure the GPIO as per usual with pinMode() and indicate with a call to attachInterrupt() the GPIO and the conditions required for the interrupt. The macro RISING indicates that isr_pulse() is to be called with the input sees a rising edge.

It is a good idea to use a pull-up resistor or configure the GPIO to use one (INPUT_PULLUP). Otherwise, the GPIO will have a floating potential when unconnected or not driven. Keep in mind that the built-in pull-up resistance is weak, somewhere between 10k and 100k (50k is typical).

Note: GPIOs 33, 34, 35, 36, and 39 are input only and do not have pull-up capability.

Frequency Counter Project

To give us some challenge and an exercise in interrupts, let's implement a frequency counter for the Wemos Lolin ESP32 using its built-in OLED.[3] Later in this chapter, versions of the same program for the TTGO ESP32 T-Display and the M5Stack are also available.

Don't despair if you lack a signal generator because the ESP32 will also generate the signal to be measured. For frequency measurement, this project makes use of the internal ESP32 pulse counter peripheral which can count pulses and interrupt under different conditions. The rising edge of the input signal is the event that will be counted.

Challenges

A frequency counter must deal with at least two challenges:

- Input signal conditioning.
- Range finding.

The input signal to the frequency counter will come from a GPIO as a 50% duty cycle PWM (Pulse Width Modulation) signal, using the PWM Arduino API. Wired to another GPIO, we are spared from having to condition the signal into the correct voltage range and make it noise-free. This approach also guarantees that the input signal will be in the correct voltage range and prevent permanent damage.

Note: Those wishing to develop this project further for general use must add input protection diodes to prevent signal excursions above 3.3 volts and protect against negative voltages. Don't rely on the weak ESD protection diodes for that job.

The main challenge in the software will be to perform the necessary automatic range finding. The pulse counter peripheral consists of a signed 16-bit counter. The peripheral counter size requires that it be configured to match the range of the incoming signal frequency. When configured incorrectly, the time between interrupts will either be too long at low frequencies or occur too frequently to be processed at high frequencies.

Approach

The strategy used in this project is to leverage the capabilities of the counter peripheral. The talents of the pulse counter include:

- A low and a high counter range can be configured.
- A threshold 0 level can be defined to cause an interrupt to occur when the counter value reaches it.
- A threshold 1 level can be defined to cause an interrupt to occur when the counter value reaches it.
- The counter can be paused at any time (including from within the ISR).
- The count can be cleared at any time.
- The counter can be resumed at any time.

The range of values for the counter used in this project are 0 to +32767 (the positive range of a signed 16-bit counter). We will measure the time that occurs between the two configured threshold values. Threshold 0 is configured with a value of +10. After clearing the counter to zero and resuming the peripheral, the first interrupt will occur shortly after. The value of threshold 1 varies according to the frequency that is being measured. A value of +32767 allows for accurate measurement but takes too long to measure a low-frequency signal. Consequently, the threshold 1 value will be low (above +10) for low-frequency measurements, and a higher value for higher frequencies.

The time between threshold interrupts is measured by using the Arduino micros() function, which returns the time in microseconds. This time is returned by the ESP32 64-bit high-resolution timer. We can now state the general frequency measurement procedure:

1. The threshold 0 value is maintained at the value +10.
2. Threshold 1 is configured with some trial value above +10.
3. The pulse counter is cleared to zero and resumed.

4. The threshold 0 interrupt occurs and the ISR captures the initial microsecond time (usec0).
5. Later the threshold 1 interrupt occurs and captures the second microsecond time, and calculates the elapsed time.
6. The pulse counter peripheral is paused to prevent too many interrupts.
7. The elapsed time in microseconds is sent by a queue to the monitoring task.
8. The monitoring task computes the frequency and displays it in the lower half of the display.

Let's check some sample measurement times. The calculation used by the monitoring task is:

$$freq = \frac{(threshold_1 - threshold_0) * 1000000}{usecs} Hz$$

where:

- freq is the frequency in Hz
- threshold0 is the value +10
- threshold1 is the trial threshold value
- usecs is the elapsed time in microseconds

Let's calculate some times so that the challenge is more evident.

Case 1 – 300,000 Hz
Plugging in the following values:

- threshold0 = +10
- threshold1 = +32767
- elapsed time = 109,190 µsec

$$300000 = \frac{(32767 - 10) * 1000000}{109190} Hz$$

The important thing to note at this point is that the measurement requires about a tenth of a second to complete (or about 109 ms). Now let's examine case 2, where the frequency has suddenly dropped to 500 Hz. Waiting for a measurement for 109 ms is quite reasonable.

Case 2 – 500 Hz
For 500 Hz, the values are:

- threshold0 = +10
- threshold1 = +32767
- elapsed time = 65.514 seconds

With the configured threshold1 of +32767, we would have to wait for 65 seconds before we could compute a frequency. This is too long to wait. This happens because the input frequency is so low, that it takes so much longer for the pulse counter peripheral to reach the two configured thresholds.

Project Code
From the two cases shown, it is clear that there is no one-size-fits all for frequency measurement. Since there is no way to know the frequency in advance, we must perform some trial measurements and zero in. The approach taken is to assume a fairly high-frequency signal and then wait for a message in the monitoring task. Listing 9-1 illustrates the Wemos Lolin ESP32 program in full.[3] The body of the monitoring task is shown below.

```
0125: static void monitor(void *arg) {
0126:   uint32_t usecs;
0127:   int16_t thres;
0128:   BaseType_t rc;
0129:
0130:   for (;;) {
0131:     rc = pcnt_counter_clear(PCNT_UNIT_0);
0132:     assert(!rc);
0133:     xQueueReset(evtq_h);
0134:     rc = pcnt_counter_resume(PCNT_UNIT_0);
0135:     assert(!rc);
0136:
0137:     rc = pcnt_get_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,&thres);
0138:     assert(!rc);
0139:     rc = xQueueReceive(evtq_h,&usecs,500);
0140:     if ( rc == pdPASS ) {
0141:       uint32_t freq = uint64_t(thres-10)
0142:           * uint64_t(1000000) / usecs;
0143:       oled_freq(freq);
0144:       thres = retarget(freq,usecs);
0145:       rc = pcnt_set_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,thres);
0146:       assert(!rc);
0147:     } else {
0148:       rc = pcnt_counter_pause(PCNT_UNIT_0);
0149:       assert(!rc);
0150:       rc = pcnt_set_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,25);
0151:       assert(!rc);
0152:     }
0153:   }
0154: }
```

Skipping the top of the loop, for now, examine line 139 where the elapsed time in microseconds is fetched from the queue. The third argument has provided a time of 500 ticks as the timeout. If the ISR does not respond with a message in 500 ticks, the else block is executed in lines 148 to 152. This block executes when the threshold1 value is too high for a quick measurement. Line 148 pauses the counter in case it has not been paused yet by the ISR. Value threshold1 is then set to a low value of +25 in line 151. This guarantees that the next measurement will be quick.

At the top of the loop, the counter is cleared (line 131), the queue is emptied (line 133), and then the pulse counter is unleashed at line 134. Line 137 fetches the current threshold1 value into the variable thres. This is cheap as it only requires a memory-mapped register read.

The execution blocks for up to 500 ticks reading from the message queue (line 139). After the ISR receives its two interrupts, it will send the monitor task a message with the elapsed microsecond time, received into variable usecs.

The block starting in line 141 is executed when the queue fetch is successful. The frequency is computed from the following values:

- thres – 10

This is the count between threshold 0 (always +10 here) and the trial threshold1 value. This is multiplied by 1000000 because the divisor is in microseconds. The resulting frequency is the computed result:

$$freq = \frac{(threshold_1 - threshold_0) * 1000000}{usecs} Hz$$

For this calculation, it is necessary to avoid overflowing the size of the 32-bit register values. Consequently, the calculation uses 64-bit values.

```
0141:       uint32_t freq = uint64_t(thres-10)
0142:           * uint64_t(1000000) / usecs;
```

Floating point is avoided for efficiency with the result in an unsigned 64-bit integer. Line 143 calls upon function oled_freq() to display the frequency in the lower half of the OLED display. Following the display of the measured frequency, the following lines attempt to find a better measurement range:

```
0144:       thres = retarget(freq,usecs);
0145:       rc = pcnt_set_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,thres);
0146:       assert(!rc);
```

A new trial threshold1 value is calculated and applied to the pulse counter peripheral in line 145.

Range Finding
At high frequencies, an improved measurement is obtained if we adjust the threshold1 value. The retarget() function tries to compute a value that will take about a tenth of a second to measure (line 91) based upon the current approximate frequency.

Two C++ lambda functions are used within this function. Don't let the C++ syntax of these scare the C-language student. Lines 94 to 99 simply define a function named target(), which is only available within the retarget() function. The frequency value freq is implicitly captured and passed into the function, so it does not need to be listed as a function argument. The single argument usec provides the measured elapsed time and the function returns a new calculated trial threshold1 value. Line 96 or 98 are executed according to the frequency range to avoid overflows. Note that the compiler determines the return type according to the returned value, which in this case is uint64_t. Lambda function useconds() computes the time needed to measure with a given threshold value.

If the calculated threshold value is greater than +3200 (line 105), then the calculation proceeds with lines 106 to 110. The procedure continues with:

1. Compute an ideal threshold value based upon the ideal measurement time (target_usecs, line 106).
2. If the calculated value is too large, then clamp the value to +32500 (line 107). Recall that the pulse counter peripheral is limited to a maximum of +32767.
3. Line 109 computes the new measurement time based upon the new threshold value.
4. A revised target value is then computed (line 110).
5. Lines 118 to 121 range check the values and clamp them to minimums/ maximums.
6. Lines 112 to 116 perform a similar downgrading of the threshold.

```
0090: static uint32_t retarget(uint32_t freq,uint32_t usec) {
0091:   static const uint32_t target_usecs = 100000;
0092:   uint64_t f = freq, t, u;
0093:
0094:   auto target = [&freq](uint32_t usec) {
0095:     if ( freq > 100000 )
0096:       return uint64_t(freq) / 1000 * usec / 1000 + 10;
0097:     else
0098:       return uint64_t(freq) * usec / 1000000 + 10;
0099:   };
0100:
0101:   auto useconds = [&freq](uint64_t t) {
0102:     return (t - 10) * 1000000 / freq;
0103:   };
0104:
0105:   if ( (t = target(usec)) > 32000 ) {
0106:     t = target(target_usecs);
```

```
    if ( t > 32500 )
      t = 32500;
    u = useconds(t);
    t = target(u);
  } else {
    t = target(target_usecs);
    if ( t < 25 )
      t = 25;
    u = useconds(t);
    t = target(u);
  }
  if ( t > 32500 )
    t = 32500;
  else if ( t < 25 )
    t = 25;
  return t;
}
```

ISR Routine

The ISR routine is relatively short and excerpted below:

```
static void IRAM_ATTR pulse_isr(void *arg) {
  static uint32_t usecs0;
  uint32_t intr_status = PCNT.int_st.val;
  uint32_t evt_status, usecs;
  BaseType_t woken = pdFALSE;

  if ( intr_status & BIT(0) ) {
    // PCNT_UNIT_0 Interrupt
    evt_status = PCNT.status_unit[0].val;
    if ( evt_status & PCNT_STATUS_THRES0_M ) {
      usecs0 = micros();
    } else if ( evt_status & PCNT_STATUS_THRES1_M ) {
      usecs = micros() - usecs0;
      xQueueSendFromISR(evtq_h,&usecs,&woken);
      pcnt_counter_pause(PCNT_UNIT_0);
    }
    PCNT.int_clr.val = BIT(0);
  }
  if ( woken ) {
    portYIELD_FROM_ISR();
  }
}
```

The ISR operates as follows:

1. The value usecs0 is declared static so that the threshold0 microsecond time will persist after the ISR returns (line 33).
2. Line 34 captures the interrupt status word for the pulse counter and this is used in line 38 to test if it applies to our pulse counter 0 unit (there are up to eight). Once it is determined that the interrupt applies to unit 0, lines 40 to 48 are executed.
3. A threshold status is obtained in line 40 for pulse counter 0.
4. Line 42 captures the microsecond timer value when the event is for threshold0 (line 41).
5. Otherwise, the elapsed time is captured in line 44 when threshold1 is received.
6. After the elapsed time is computed, it is sent by message queue in line 45.
7. Immediately afterwards, the counter is paused in line 46. This prevents the ESP32 from being hammered with interrupts should the threshold value range be too small for the frequency being measured.
8. Finally, the interrupt flag is cleared in line 46.

Note: the microsecond timer can overflow. Correcting this is left as an exercise for the reader.

xQueueSendFromISR()
Two important things from the ISR code should be noted, where the value was queued (line 45):

1. Function xQueueSendFromISR() was used instead of xQueueSendToBack().
2. Argument three of xQueueSendFromISR() was a pointer to a BaseType_t woken value instead of supplying a timeout value.

ISR code must always use the FreeRTOS function using the suffix "FromISR". These functions are specially designed to be used from an ISR. The normal FreeRTOS functions must not be used.

There is no timeout argument to the ISR specific xQueueSendFromISR() call. This is because the function is not permitted to block within an ISR routine. If the queue is full, the call will immediately fail rather than block.

Note: xQueueSendFromISR() does not accept a timeout parameter because the call is not permitted to block within an ISR. If the queue is full, the return value will represent an immediate fail. This may have consequences for the design of your application.

The third argument of xQueueSendFromISR() is instead a pointer to a value named woken (declared in line 36). This value can be supplied as a nullptr/NULL, if you don't need the value returned. The idea behind the argument is to signal when to invoke the scheduler upon ISR return. In this example, if queuing an item to the queue wakes up an equal or higher priority task than the interrupted task, then it is necessary to call the scheduler. Invoking the scheduler allows it to select that unblocked task to run.

If there were no tasks unblocked with equal or higher priority, then there is no need to call the scheduler, causing the returned value of woken to be zero. This saves execution time in the ISR. Lines 50 and 51 illustrate how the scheduler is conditionally invoked.

portYIELD_FROM_ISR()
What happens if the woken argument is supplied as nullptr, or otherwise ignored? In other words, what happens if portYIELD_FROM_ISR() is *not* called when the value returned is true (non-zero)? The result is that if the flag is ignored, the currently executing task will always resume when the ISR returns. This will be true even when the FreeRTOS call unblocked an equal or higher priority task. This priority problem will *eventually* be corrected at the next system tick, another blocking call from a task or interrupt that invokes the scheduler. Otherwise ignoring the flag permits a lower priority task to resume when a higher priority task should be executing instead.

This begs another question – why make the woken argument optional? In a given program it may be that no higher priority tasks are running affected. This still affects equal priority tasks. However, this might be less critical. Another potential reason to ignore the woken flag is when the ISR is invoked with high frequency. To keep the ISR execution time as short as possible, then it is probably best to skip the call into the scheduler.

The call to portYIELD_FROM_ISR() should be performed as the last executed step of the ISR routine if it is invoked at all. Some embedded platforms may not return from this call, while others may. The code should be designed to operate either way.

Running the Demo
Figure 9-1 is the wiring diagram for the frequency counter project using the Wemos Lolin ESP32, with built-in OLED (SSD1306). The GPIO values can be customized at the top of the program if required, but the schematic provides a recommended configuration. The 10 Kohm pot R_1 provides the analog voltage into GPIO 14. This selects a frequency generated PWM output frequency on GPIO 25. The frequency counter measures the frequency on input GPIO 26. On the breadboard, simply jumper a wire from GPIO 25 to 26. No Serial Monitor is used, so the project will run independently.

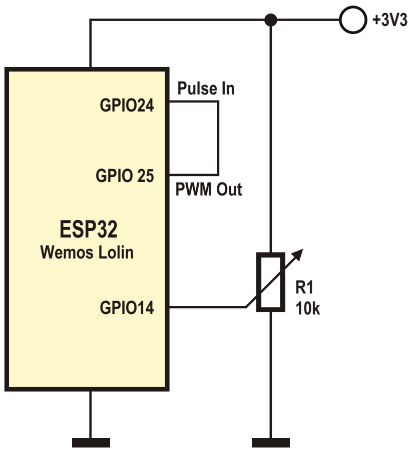


Figure 9-1. Schematic of the Wemos Lolin ESP32 project for the Frequency Counter. Note that the GPIOs differ for the TTGO and M5Stack (see text).

Figure 9-2 illustrates the Wemos Lolin ESP32 breadboarded with the potentiometer and the jumper wire configured. When powered by the USB cable, the OLED display should spring to life with the PWM generated frequency displayed in the upper half of the display. The bottom half will often lag but when the frequency is held stable, the counter will zero in on the correct frequency.

The ADC input value, with its voltage controlled by the pot R_1, has a range of 0 to 4095 (12-bits). This is multiplied by 80 and 500 added to it to compute the desired frequency (Line 244). Consequently, the generated PWM frequency ranges from 500 Hz to about 328.1 kHz. Instability in the ADC readings will cause the generated frequency to jump around somewhat. This results in the frequency counter readings jumping around as it tries to keep up.

With the pot correctly wired, a fully counter-clockwise turn should cause a generated signal of 500 Hz. The OLED should display "500 gen" at the top of the display. The frequency counter value is displayed in the bottom half and should eventually settle as "500 Hz". Crank the pot to the mid-range, and the frequency counter should catch up and match it fairly close, if not exactly. Crank the pot fully clockwise, and the signal near 328 kHz should be generated, with the frequency counter matching closely. Try various settings in between. Because the ADC reading is multiplied by 80, the frequency selection is rather coarse.

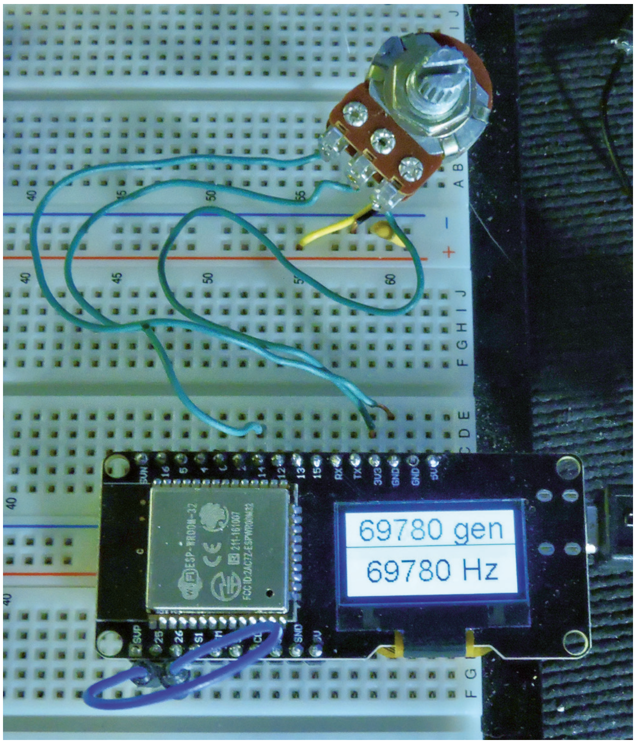


Figure 9-2. The Wemos Lolin ESP32 with OLED breadboarded as a Frequency Counter.

Troubleshooting the Wemos Lolin ESP32

1. If the display remains dark, or otherwise scrambled, then make sure that you have the correct driver installed. Review Chapter 1, to make certain the correct OLED display driver has been installed.

2. If the display works, but the frequency counter value is not being updated, check the frequency counter input (GPIO 25). Make certain that the PWM output (on GPIO 26) is jumpered to GPIO 25.

3. If the displayed PWM frequency is always high (in the 200-300 kHz range), check that you have wired the potentiometer correctly. The outer legs of the pot should go to +3.3 volts and ground. The middle tab (wiper arm) should be wired to GPIO 14. Be certain not to wire anything to +5 volts.

Note: The frequency counter takes longer to lock onto a low-frequency signal than it does for a high-frequency signal.

Pulse Counter Notes

The focus of this chapter has been upon the FreeRTOS interaction with ISR code. The keen reader might question the call to pcnt_counter_pause() from within the ISR code (line 46).

He/she is wise to question this because the ISR must be quick and never have reason to block. Sometimes the developer has to get their hands dirty and just examine the source code being called.

Another clue to its safety is found in the ESP Technical Reference Manual. An examination of the pulse counter peripheral chapter reveals that the counter is paused by setting a bit in a memory-mapped register. This is something quite safely performed from an ISR.

On the other hand, other peripherals may not be so simply controlled. It is often unwise to invoke peripheral control functions from within an ISR. So it’s fair to ask, how do you know? Unfortunately, there is no universal rule beyond examining the source code. Just be aware of this so that your "spidey senses" will tingle when you identify similar situations.

For those wishing to learn more about the pulse counter peripheral found inside the ESP32, you can find documentation at this resource.[4]

Setup for Interrupts

After the pulse counter peripheral is configured in the function counter_init(), you will find a few statements related to interrupts. Line 203 configures the interrupt handler to be used while line 205 enables the interrupts. When writing code like this be certain that the handler is ready to accept interrupts after they are enabled because sometimes they can occur immediately. In this case, the handle to the message queue must be ready.

```
0203:   rc = pcnt_isr_register(pulse_isr,nullptr,0,&isr_handle);
0204:   assert(!rc);
0205:   rc = pcnt_intr_enable(PCNT_UNIT_0);
0206:   assert(!rc);
```

Interrupt setup is often performed along these lines. Generally, the peripheral must be readied first, then configure the interrupt vector for the interrupt handler, and then enable the peripheral to raise interrupts.

```
0001: // freqctr.ino
0002:
0003: #define GPIO_PULSEIN  25
0004: #define GPIO_FREQGEN  26
0005: #define GPIO_ADC      14
0006:
0007: // GPIO for PWM output
0008: #define PWM_GPIO      GPIO_FREQGEN
0009: #define PWM_CH        0
0010: #define PWM_FREQ      2000
0011: #define PWM_RES       1
0012:
0013: #include "SSD1306.h"
0014: #include "driver/periph_ctrl.h"
```

```
#include "driver/pcnt.h"

#define SSD1306_ADDR  0x3C
#define SSD1306_SDA   5
#define SSD1306_SCL   4

static SSD1306 oled(
  SSD1306_ADDR,
  SSD1306_SDA,
  SSD1306_SCL
);

static int app_cpu = 0;
static pcnt_isr_handle_t isr_handle = nullptr;
static SemaphoreHandle_t sem_h;
static QueueHandle_t evtq_h;

static void IRAM_ATTR pulse_isr(void *arg) {
  static uint32_t usecs0;
  uint32_t intr_status = PCNT.int_st.val;
  uint32_t evt_status, usecs;
  BaseType_t woken = pdFALSE;

  if ( intr_status & BIT(0) ) {
    // PCNT_UNIT_0 Interrupt
    evt_status = PCNT.status_unit[0].val;
    if ( evt_status & PCNT_STATUS_THRES0_M ) {
      usecs0 = micros();
    } else if ( evt_status & PCNT_STATUS_THRES1_M ) {
      usecs = micros() - usecs0;
      xQueueSendFromISR(evtq_h,&usecs,&woken);
      pcnt_counter_pause(PCNT_UNIT_0);
    }
    PCNT.int_clr.val = BIT(0);
  }
  if ( woken ) {
    portYIELD_FROM_ISR();
  }
}

static void lock_oled() {
  xSemaphoreTake(sem_h,portMAX_DELAY);
}

static void unlock_oled() {
  xSemaphoreGive(sem_h);
```

```
}

static void oled_gen(uint32_t f) {
  char buf[32];

  snprintf(buf,sizeof buf,"%u gen",f);
  lock_oled();
  oled.setColor(BLACK);
  oled.fillRect(0,0,128,31);
  oled.setColor(WHITE);
  oled.drawString(64,0,buf);
  oled.drawHorizontalLine(0,26,128);
  oled.display();
  unlock_oled();
}

static void oled_freq(uint32_t f) {
  char buf[32];

  snprintf(buf,sizeof buf,"%u Hz",f);
  lock_oled();
  oled.setColor(BLACK);
  oled.fillRect(0,32,128,63);
  oled.setColor(WHITE);
  oled.drawString(64,32,buf);
  oled.display();
  unlock_oled();
}

static uint32_t retarget(uint32_t freq,uint32_t usec) {
  static const uint32_t target_usecs = 100000;
  uint64_t f = freq, t, u;

  auto target = [&freq](uint32_t usec) {
    if ( freq > 100000 )
      return uint64_t(freq) / 1000 * usec / 1000 + 10;
    else
      return uint64_t(freq) * usec / 1000000 + 10;
  };

  auto useconds = [&freq](uint64_t t) {
    return (t - 10) * 1000000 / freq;
  };

  if ( (t = target(usec)) > 32000 ) {
    t = target(target_usecs);
    if ( t > 32500 )
      t = 32500;
    u = useconds(t);
    t = target(u);
  } else {
    t = target(target_usecs);
    if ( t < 25 )
      t = 25;
    u = useconds(t);
    t = target(u);
  }
  if ( t > 32500 )
    t = 32500;
  else if ( t < 25 )
    t = 25;
  return t;
}

static void monitor(void *arg) {
  uint32_t usecs;
  int16_t thres;
  BaseType_t rc;

  for (;;) {
    rc = pcnt_counter_clear(PCNT_UNIT_0);
    assert(!rc);
    xQueueReset(evtq_h);
    rc = pcnt_counter_resume(PCNT_UNIT_0);
    assert(!rc);

    rc = pcnt_get_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,&thres);
    assert(!rc);
    rc = xQueueReceive(evtq_h,&usecs,500);
    if ( rc == pdPASS ) {
      uint32_t freq = uint64_t(thres-10)
          * uint64_t(1000000) / usecs;
      oled_freq(freq);
      thres = retarget(freq,usecs);
      rc = pcnt_set_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,thres);
      assert(!rc);
    } else {
      rc = pcnt_counter_pause(PCNT_UNIT_0);
      assert(!rc);
      rc = pcnt_set_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,25);
      assert(!rc);
    }
  }
}

static void oled_init() {
  oled.init();
  oled.clear();
  oled.flipScreenVertically();
  oled.invertDisplay();
  oled.setTextAlignment(TEXT_ALIGN_CENTER);
  oled.setFont(ArialMT_Plain_24);
  oled.drawString(64,0,"freqctr.ino");
  oled.drawHorizontalLine(0,0,128);
  oled.drawHorizontalLine(0,26,128);
  oled.display();
}

static void analog_init() {
  adcAttachPin(GPIO_ADC);
  analogReadResolution(12);
  analogSetPinAttenuation(GPIO_ADC,ADC_11db);
}

static void counter_init() {
  pcnt_config_t cfg;
  int rc;

  memset(&cfg,0,sizeof cfg);
  cfg.pulse_gpio_num = GPIO_PULSEIN;
  cfg.ctrl_gpio_num = PCNT_PIN_NOT_USED;
  cfg.channel = PCNT_CHANNEL_0;
  cfg.unit = PCNT_UNIT_0;
  cfg.pos_mode = PCNT_COUNT_INC; // Count up on the positive edge
  cfg.neg_mode = PCNT_COUNT_DIS;
  cfg.lctrl_mode = PCNT_MODE_KEEP;
  cfg.hctrl_mode = PCNT_MODE_KEEP;
  cfg.counter_h_lim = 32767;
  cfg.counter_l_lim = 0;
  rc = pcnt_unit_config(&cfg);
  assert(!rc);

  rc = pcnt_set_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_0,10);
  assert(!rc);
  rc = pcnt_set_event_value(PCNT_UNIT_0,PCNT_EVT_THRES_1,10000);
  assert(!rc);
  rc = pcnt_event_enable(PCNT_UNIT_0,PCNT_EVT_THRES_0);
  assert(!rc);
  rc = pcnt_event_enable(PCNT_UNIT_0,PCNT_EVT_THRES_1);
  assert(!rc);
  rc = pcnt_counter_pause(PCNT_UNIT_0);
  assert(!rc);
  rc = pcnt_isr_register(pulse_isr,nullptr,0,&isr_handle);
  assert(!rc);
  rc = pcnt_intr_enable(PCNT_UNIT_0);
  assert(!rc);
}

void setup() {
  unsigned ms;
  BaseType_t rc;  // Return code

  app_cpu = xPortGetCoreID();
  sem_h = xSemaphoreCreateMutex();
  assert(sem_h);
  evtq_h = xQueueCreate(20,sizeof(uint32_t));
  assert(evtq_h);

  // Use PWM to drive CLKIN
  ledcSetup(PWM_CH,PWM_FREQ,PWM_RES);
  ledcAttachPin(PWM_GPIO,PWM_CH);
  ledcWrite(PWM_CH,1); // 50%

  counter_init();
  oled_init();
  delay(2000);

  // Start the monitor task
  rc = xTaskCreatePinnedToCore(
    monitor,    // Function
    "monitor",  // Name
    4096,       // Stack size
    nullptr,    // Argument
    1,          // Priority
    nullptr,    // Handle ptr
    app_cpu     // CPU
  );
  assert(rc == pdPASS);
}

void loop() {
  uint32_t f; // Frequency

  f = analogRead(GPIO_ADC) * 80 + 500;
  oled_gen(f);

  ledcSetup(PWM_CH,f,PWM_RES);
  ledcAttachPin(PWM_GPIO,PWM_CH);
  ledcWrite(PWM_CH,1); // 50%
  delay(500);
}
```

Listing 9-1.Frequency counting program freqctr.ino for the Lolin ESP32 with OLED.

TTGO ESP32 T-Display

For those that have the TTGO ESP32 T-Display unit, the demonstration program is available for it in color.[5] This program is the same as the Wemos Lolin program, except that the graphics driver support differs and the ADC is configured for GPIO 15 instead (note the GPIO differences here when viewing Figure 9-1).

The GPIO declarations are shown below and notice the include of the TFT_eSPI.h header file:

```
// freqctr.ino - TTGO ESP32 T-Display

#define GPIO_PULSEIN  25
#define GPIO_FREQGEN  26
#define GPIO_ADC      15

// GPIO for PWM output
#define PWM_GPIO      GPIO_FREQGEN
#define PWM_CH        0
#define PWM_FREQ      2000
#define PWM_RES       1

#include <SPI.h>
#include <TFT_eSPI.h>
```

The driver can be downloaded through the Arduino menu Tools -> Manage Libraries. Search for "TFT_eSPI" for the library authored by Bodmer. I tested with version 2.1.4 at the time of writing. Install that version or a newer one, assuming compatibility has been maintained. But don’t stop there!

This driver requires that you to edit one of the header files, once it has been installed. Change to the appropriate directory for your platform (Table 9-1). Then edit the header file named User_Setup_Select.h and make the following changes:

1. Comment out the include file for User_Setup.h. Just prefix that line with "//" to comment it out:

```
// #include <User_Setup.h>    // Default setup is root library folder
```

2. Uncomment the following line by removing the inital "//" (approximately line 53):

```
#include <User_Setups/Setup25_TTGO_T_Display.h> // Setup file for
ESP32 and TTGO..
```

The last statement will configure the driver for the display that the TTGO ESP32 T-Display uses. Save the changes to User_Setup_Select.h. This will complete your driver configuration.

Directory Pathname	**Platform**
~/Documents/Arduino/libraries/TFT_eSPI	Mac
C:\Users\<user name>\Documents\Arduino\libraries\TFT_eSPI	Windows

Table 9-1. Locations for the User_Setup_Select.h file.

Return to your Arduino IDE and recompile and upload the binary. Once the device resets, a green screen should appear with the text "freqctr-ttgo.ino". After two seconds, the PWM Generator and Frequency Counter display should indicate on the upper and lower halves of the landscape-oriented display.

Troubleshooting the TTGO

1. If the display remains dark, or otherwise scrambled, then recheck your driver install. You must make the recommended changes to the installed header file. Recompile and upload after any changes made.

2. If the display works, but the lower half of the display shows red, this means that the program is not seeing pulses on the frequency counter input (GPIO 25). Recheck your wiring. Make certain that the PWM output (on GPIO 26 in this example) is jumpered to GPIO 25.

3. If the displayed PWM frequency is always high (in the 200-300 kHz range), check that you have wired the potentiometer correctly. The outer legs of the pot should go to +3.3 volts and ground. The middle tab (wiper arm) should be wired to GPIO 15. Be certain not to wire anything to +5 volts.

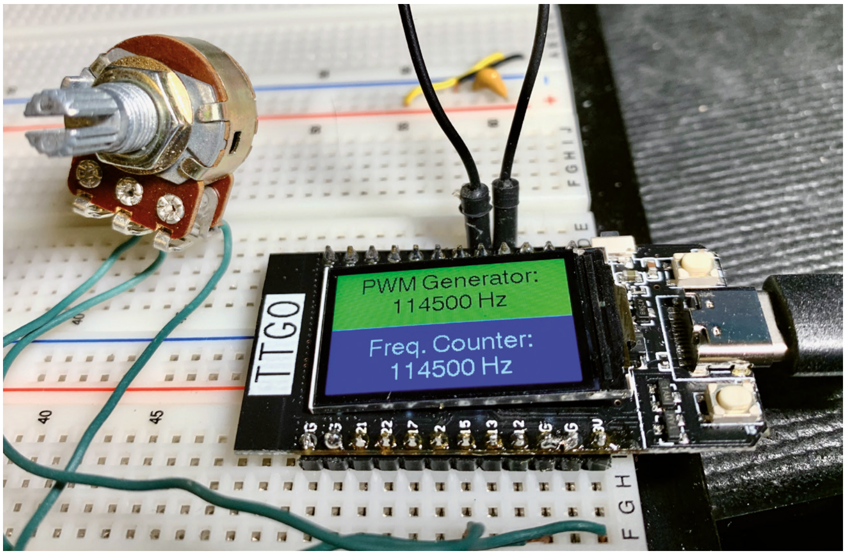


Figure 9-3. The TTGO ESP32 T-Display running the freqctr-ttgo.ino program.

M5Stack

For even more fun, M5Stack owners can use the demo program freqctr-m5.ino[6]. This program uses different GPIO assignments due to pin usage within the M5Stack. These are reflected in the initial few lines of the program. Note that these GPIO assignments differ from the ones shown in Figure 9-1.

```
// freqctr.ino - M5Stack

#define GPIO_PULSEIN  35
#define GPIO_FREQGEN  2
#define GPIO_ADC      36

// GPIO for PWM output
#define PWM_GPIO      GPIO_FREQGEN
#define PWM_CH        0
#define PWM_FREQ      2000
#define PWM_RES       1

#include <M5Stack.h>
```

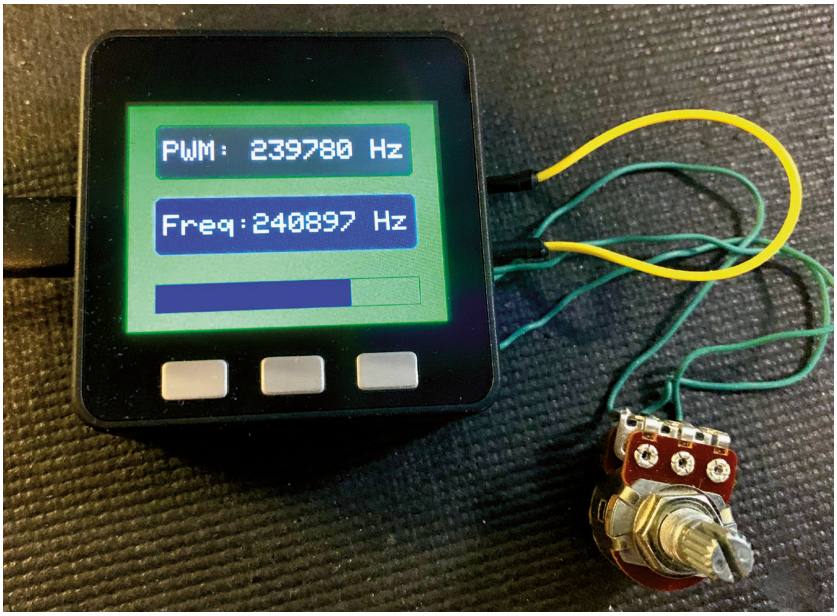


Figure 9-4. The M5Stack running demo freqctr-m5.ino.

The olive-colored box (top) displays the generated PWM output frequency. The Blue box in the middle displays the measured frequency (Figure 9-4 shows that the program is catching up to a changed frequency). The progress bar at the bottom displays the measured ADC value from the pot, read from GPIO 36.

The unit can be powered off by pressing Button A (lower left) on the M5Stack. Pushing it again will power the device on again.

Troubleshooting M5Stack

The M5Stack should give very little trouble after the appropriate API library is installed (revisit Chapter 1 if there is an issue compiling).

1. Do not be concerned about a fairly audible click when the unit starts. This is due to the silencing of the speaker on the M5Stack. Unfortunately, you may still hear a slight noise when the unit is powered off.

2. If the display remains dark, or otherwise scrambled, then recheck the driver install. Always recompile and upload after driver and library changes.

3. If the frequency portion of the display is missing (in blue), it is likely that the program is not receiving pulses on the frequency counter input (GPIO 35). Recheck the wiring. Make certain that the PWM output (on GPIO 2 in this example) is jumpered to GPIO 35.

4. If the displayed PWM frequency is always high (in the 200-300 kHz range), check that you have wired the potentiometer correctly. The outer legs of the pot should go to +3.3 volts and ground. The middle tab (wiper arm) should be wired to GPIO 36. Be certain *not to wire anything to +5 volts.*

Summary

This chapter has described the various ways that ISR routines are special. Within them, only FreeRTOS functions with names ending in FromISR should be used. ISRs must be short in execution and never block. There were other restrictions for calling non-FreeRTOS functions that must be observed. In the ESP32, the ISR itself needs the IRAM_ATTR macro in its declaration. Yet despite all of the restrictions discussed in this chapter, interrupts serve as an effective conduit into applications when properly applied.

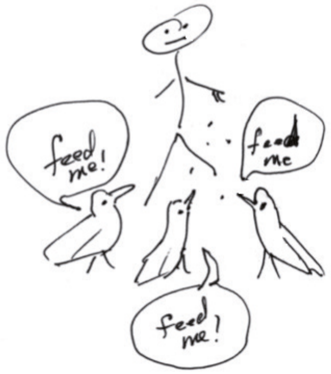
Exercises

1. Why must a function be recursive to be called from within an ISR?
2. Why is it a bad idea to call routines like printf, snprintf() etc.?
3. Name the reason why malloc() or free() should never be called from an ISR.
4. What stack does the ISR stack frame get allocated from?
5. What factors reduce the maximum stack space available for an ISR?
6. Why is there no timeout parameter for calls like xQueueSendFromISR()?
7. What is the purpose of the woken (third) argument to the function xQueueSend-FromISR()?
8. How is the task scheduler invoked from within an ISR?
9. What happens if the task scheduler is not invoked when the woken (third) argument to xQueueSendFromISR() is returned with the value pdTRUE?
10. What three events cause the FreeRTOS task scheduler to be called?
11. Does the call to macro portYIELD_FROM_ISR() return?
12. Why is it safe to invoke the pulse counter peripheral routine pcnt_counter_pause() from within the ISR?
13. What happens when the queue is full when calling xQueueSendFromISR()?
14. Why should the macro portYIELD_FROM_ISR() be coded as the last statement executed in the ISR?
15. Is it permissible to call delay() from within an ISR?

Web Resources

[1] https://sourceware.org/newlib/
[2] https://esp32.com/viewtopic.php?t=5111
[3] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/freqctr/freqctr.ino
[4] https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/pcnt.html
[5] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/freqctr-ttgo/freqctr-ttgo.ino
[6] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/freqctr-m5/freqctr-m5.ino

Chapter 10 • Queue Sets



I feel like I am serving multiple queues.

As you write increasingly complex embedded applications, you may find that sometimes you need to block on several FreeRTOS resources from one particular task. The task needs a way to suspend its execution until an event occurs in any of a set of queues or semaphores of interest. FreeRTOS provides the solution for this in the form of the queue set.

The Problem

Let's begin by stating the problem example in more concrete terms. You've been tasked to write a program that takes GPIO inputs from three push buttons. When the button is pressed, the LED goes off and lit after the button is released (in addition to triggering some game events). Because this is for gaming, multiple buttons can be pressed simultaneously. To be as responsive to button events as possible, interrupts will be used to sense the GPIO signal changes.

As the designer, you might initially allocate one queue to post all button events to. The queue would carry the button press/release events. But button contacts can bounce, especially the ones made with metal contacts. A single queue might get filled bounce events from one button just when the game player wants to activate his/her smart bomb from another button. Since the ISR will be queuing the button events, the event will be lost when the queue becomes full (recall that the ISR cannot block when the queue is full). Is there a better solution to this problem?

To prevent the queue from becoming full because of another button's stuttering, you decide that each button needs its own message queue. In this way, if a problematic button bounces too much, it will only loose events for that button alone. This design now leaves you with three message queues for input processing.

The input event processing task is a single task. Ideally, at the top of the loop, the task would block until an event is received from any of those three queues. But the call to xQueueReceive() can only receive from one queue at a time. As a work-around, you might use a zero timeout poll of all three queues but after that, it is desirable to pause the execution if there is nothing further to process. This leaves more CPU time available to the rest of the gaming code.

The Queue Set

To solve the problem of blocking on multiple queues, the FreeRTOS queue set comes to the rescue. The first step is to create a queue set resource, where a queue set handle is returned:

```
QueueSetHandle_t qsh;

qsh = xQueueCreateSet(const UBaseType_t uxEventQueueLength);
```

The single input parameter is an event queue length. To prevent possible loss of events, this queue length must be the sum of all queue depths used. For binary semaphores and mutexes add one for each and for counting semaphores add count entries.

Queue Set Configuration

Before the queue set can be useful however, the participating queues, binary semaphores, and mutexes must be added to the queue set:

```
BaseType_t rc;

rc = xQueueAddToSet(QueueSetMemberHandle_t mh,QueueSetHandle_t qsh);
assert(rc == pdPASS);
```

The call to xQueueAddToSet() configures the queue set to monitor the specified queue, semaphore or mutex. The resource handle to be added is provided in the *first* argument, with the queue set handle as the second parameter.

Note: It is easy to get the argument order for xQueueAddToSet() incorrect. The queue set argument is the *second* argument, while the resource being added is the first argument.

Queue Set Select

Once you have a configured queue set, the queue set can be used to block execution until any one of the resources becomes active:

```
QueueSetMemberHandle_t mh;

mh = xQueueSelectFromSet(QueueSetHandle_t hqs,const TickType_t
xTicksToWait);
```

When there is no timeout, the xQueueSelectFromSet() call returns a handle to a newly activated resource. If the request timed out, then the returned member handle mh will be nullptr (or NULL). It is left to the caller to match the returned handle mh with the resource that is a member of the queue set. Once the caller has determined the source of the event, then the appropriate queue/semaphore/mutex operation can be performed. For example, if mh returned the handle of a *queue,* the caller would now fetch from the queue with xQueueReceive():

```
        BaseType_h rc;
        item_t item;

        rc = xQueueReceive(mh,&item,0);
        assert(rc == pdPASS);
```

The example used a timeout of zero because the xQueueSelectFromSet() has already informed that this queue has an item available. If not, there is a bug (the assert() macro should raise the alarm).

Queue Set Traps To Avoid
There are two traps to avoid for those using FreeRTOS queue sets. Both of these can frustrate and cost you debugging time.

xQueueAddToSet Trap 1
This first trap was briefly mentioned as part of configuring the queue set. The problem is the order of the arguments in the call to xQueueAddToSet():

```
        BaseType_t xQueueAddToSet(
          QueueSetMemberHandle_t xQueueOrSemaphore,
          QueueSetHandle_t xQueueSet
        );
```

Normally the resource being operated on is listed as the first argument in FreeRTOS. But the xQueueAddToSet() breaks the mold by listing it second. The handle being added to the queue is supplied as the *first* argument. The compiler used by the ESP32 Arduino framework that I was using, didn't raise a warning when I reversed the two arguments. The following assertion caught the problem but it cost me some head-scratching to determine why. So repeat after me "The *queue set* handle is the second argument of the xQueueAddToSet() call". Remembering that will speed your code development.

xQueueAddToSet Trap 2
The second trap is less obvious – even when the argument order is specified correctly and the handles are valid, the call xQueueAddToSet() *can still return pdFAIL.* Discovering why may take some time.

It is considered a best practice to declare and use program elements in the scope where they will be used. The idea applied was to create and configure the queue set in the using task. No other element of the code references this queue set.

The problem occurred because an interrupt was involved. The resources were initialized and the ISR was established in the setup() function. By the time the queue set using task was running and attempting to configure the queue set, an interrupt had already been processed and queued an event to one of the member queues. The function xQueueAddToSet() will not add a resource like a *given* binary semaphore or a queue that is not empty. Similar restrictions for mutex exist. The lesson here is that the queue set must have its members added to the set before they become active.

Demonstration

The demonstration for the queue set in this chapter has been boiled down to its bare essentials so that you can focus on the mechanism rather than the code. Like the problem statement given earlier, this demo controls three LEDs based upon three input buttons. Each button has a change sensed by interrupt and handled by an ISR. Each button ISR possesses its own queue that the ISR can queue a button press/release event to. The program has been written so that when a button is not pressed, the LED is lit. When you start the program, all three LEDs should be lit. If you don't see this then recheck the wiring or LED polarity. Additionally, you can hold down one, two, or three buttons at a time. For example, holding two buttons down should darken the two associated LEDs.

Figure 10-1 illustrates the wiring used by this demonstration, which can be run on any dev board with enough GPIOs. The Serial Monitor is not used. Each of the three LEDs are sourced by their respective GPIO pads through a 220-ohm current limiting resistor. The buttons have a common ground and are wired to their respective GPIO inputs.

Note: Even though GPIO pins 33 and higher could have been used as inputs for this project, they were avoided because they lack pull-up resistances that are required. These GPIOs can be used however if an external pull-up resistor is added to the circuit with a value of 10 k to 50 Kohms.

The push buttons for this example (Figure 10-2) use a small PCB with four buttons sharing a common ground (this demo only used three of the four buttons). You can, of course, use separate push buttons if you wish.

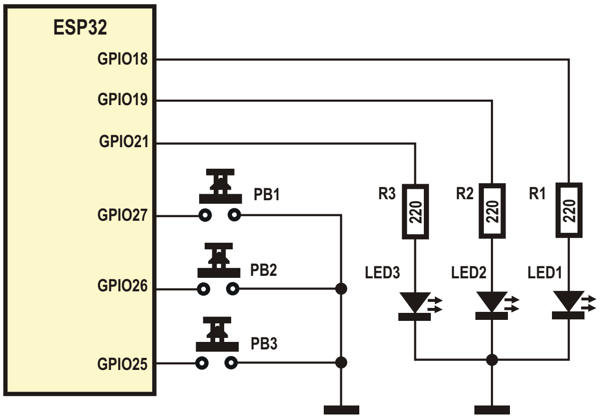


Figure 10-1. The demonstration schematic for the program qset.ino.

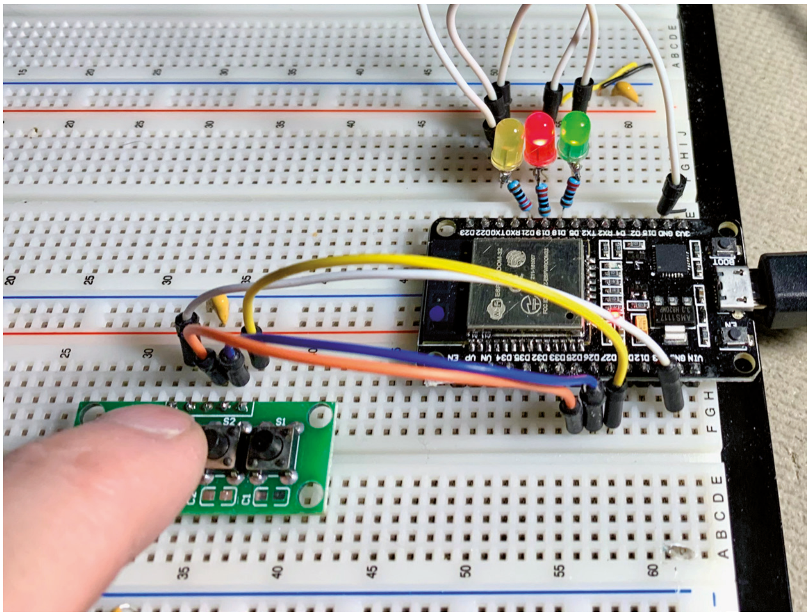


Figure 10-2. Breadboarded qset.ino demonstration with one button pressed using a generic dev board.

Program Breakdown

To reduce the amount of code necessary for this demonstration, the three buttons are configured by means of the s_button structure:

```
0023: static struct s_button {
0024:   int           butn_gpio;  // Button
0025:   int           led_gpio;   // LED
0026:   QueueHandle_t qh;         // Queue
0027:   isr_t         isr;        // ISR routine
0028: } buttons[N_BUTTONS]
```

The structure members butn_gpio and led_gpio are initialized with the GPIO numbers used by the paired button and LED. For example, the first structure is initialized with:

```
0029:   { GPIO_BUT1, GPIO_LED1, nullptr, isr_gpio1 },
```

The value qh (line 26) is the queue handle to be used for the ISR routine to queue the event to, and will be initialized within the setup() routine. Finally, the isr member is a function pointer to the routine that should be used for the input GPIO interrupt processing.

setup()
The setup() function performs the following critical initialization:

```
0094:   QueueSetHandle_t hqset;
0095:   BaseType_t rc;
0096:
0097:   hqset = xQueueCreateSet(Q_DEPTH*N_BUTTONS);
0098:   assert(hqset);
0099:
0100:   // For each button + LED pair:
0101:   for ( unsigned ux=0; ux<N_BUTTONS; ++ux ) {
0102:     s_button& button = buttons[ux];
0103:
0104:     button.qh = xQueueCreate(Q_DEPTH,sizeof(bool));
0105:     assert(button.qh);
0106:     rc = xQueueAddToSet(button.qh,hqset);
0107:     assert(rc == pdPASS);
0108:     pinMode(button.led_gpio,OUTPUT);
0109:     digitalWrite(button.led_gpio,1);
0110:     pinMode(button.butn_gpio,INPUT_PULLUP);
0111:     attachInterrupt(button.butn_gpio,button.isr,CHANGE);
0112:   }
```

The queue set is created before any of the queues are created (line 97). The loop in lines 101 to 112 then configure for each LED and button pair.

Line 102 assigns a C++ structure *reference* for convenience and code cleanliness. For C programmers, that are not used to this, just realize that this statement makes referring to buttons[ux] more convenient within the loop by just naming button. Don't be concerned about the efficiency as the compiler knows how to eliminate unnecessary pointers and references. Note also that it is not necessary in line 102 to state that this is a struct. Structures and classes in C++ are already in the same type namespace as other types, unlike the C language.

The button specific queue is created in line 104 and then added to the queue set in line 106. This happens before the ISR handler is established, so this should never fail (line 107). Lines 108 and 109 configure the output GPIO for driving the LED and establish an initial state of *on*. So if after flashing this demo you don't see the LEDs lit, recheck the wiring, and the polarity of the LED.

Lines 110 and 111, establish the button input GPIO, and configure the ISR to receive calls when the GPIO signal rises or falls (CHANGE).

ISR Routines
The Arduino environment does not make it easy to share one ISR routine for three different GPIO inputs. The ISR needs to know which GPIO changed state (rising or falling). With some advance fiddling, you could do that using the ESP32 hardware registers but let's save that for another day. In this demonstration, the problem was addressed by using three different ISR routines, customized by their GPIO index (into array buttons[]). For example, the ISR for button 1 is:

```
0069: // ISR specific to Button 1
0070:
0071: static void IRAM_ATTR isr_gpio1() {
0072:   if ( isr_gpiox(0) )
0073:     portYIELD_FROM_ISR();
0074: }
```

This code in turn, invokes an inlined function to do all of the dirty work:

```
0058: // Generalized ISR for each GPIO
0059:
0060: inline static BaseType_t IRAM_ATTR isr_gpiox(uint8_t gpiox) {
0061:   s_button& button = buttons[gpiox];
0062:   bool state = digitalRead(button.butn_gpio);
0063:   BaseType_t woken = pdFALSE;
0064:
0065:   (void)xQueueSendToBackFromISR(button.qh,&state,&woken);
0066:   return woken;
0067: }
```

For new students, the *inline* keyword is important in this context, since the ISR needs to be short as we can make it. The *inline* keyword indicates to the compiler that it should not generate a separate function to be called (requiring save/restore of registers), but rather to code the statements inline where it is invoked. So line 72 above will expand into the code shown in lines 60 to 67. In the old days of C, you might have used a macro to do this. This is far cleaner and expressive.

The new student might be inclined to paste this code into three places. But the inlined function permits us to code the routine in one place. All three ISRs only vary by the index into the buttons array. If an adjustment is required to the ISR code, the adjustment can be done in one place. This is the kind of practice that professionals use to ease code maintenance. It also involves less code to read. Win/win.

Line 72 checks to see if the woken argument was set to true. If queuing an entry wakes up a higher or equal priority task, then the scheduler is invoked before returning.

Event Monitoring Task

Now let's examine the star of this demonstration – the evtask() monitoring task. The queue set that is created in the setup() function (line 97) is passed as an argument to the task (line 119), and recast in line 37. Passing it as an argument prevents any other section of code from having access to this handle. Only one task should be receiving from a queue set since there is a race condition between having the handle selected (line 44) and receiving from the queue (line 49). Keeping the queue set in a restricted scope eliminates this risk. In a small project like this, the student may think this technique is pedantic but in larger industrial projects you'll be glad for it.

The xQueueSelectFromSet() is called in line 44 at the top of the evtask() forever loop. Here it will block forever until one of the three queues receives an event (populated by one of the ISR routines). When the function returns, we are guaranteed to have a handle to one of the three queues. Using the queue's handle, we search for it in the array buttons[] (lines 45 to 54) so that we can determine which LED to change. The affected LED is written to by line 51 after the queue event is received in line 49.

```
0036: static void evtask(void *arg) {
0037:   QueueSetHandle_t hqset = (QueueSetHandle_t*)arg;
0038:   QueueSetMemberHandle_t mh;
0039:   bool bstate;
0040:   BaseType_t rc;
0041:
0042:   for (;;) {
0043:     // Wait for an event from our 3 queues:
0044:     mh = xQueueSelectFromSet(hqset,portMAX_DELAY);
0045:     for ( unsigned ux=0; ux<N_BUTTONS; ++ux ) {
0046:       s_button& button = buttons[ux];
0047:
0048:       if ( mh == button.qh ) {
0049:         rc = xQueueReceive(mh,&bstate,0);
0050:         assert(rc == pdPASS);
0051:         digitalWrite(button.led_gpio,bstate);
0052:         break;
0053:       }
0054:     }
0055:   }
0056: }
```

That is essentially the magic of queue sets. Let's restate the summary of the procedure:

1. Create the queue set with xQueueCreateSet().
2. Create the necessary queue, semaphore, or mutex.
3. Add the queue, semaphore, or mutex to the queue set.
4. Repeat steps 2 and 3 until all required resources are configured for the queue set.

5. In the event loop, call xQueueSelectFromSet() to wait for an event from any resource in the set.
6. Call the appropriate resource API for the member handle returned (xQueueReceive() for queues, for example).

Mutexes
There is one problem related to mutexes when using queue sets. There is *no priority boost* from a higher priority task blocking on xQueueSelectFromSet() for the lower priority task owning the mutex, in the queue set. If priority inversion must be avoided at all costs, then queue sets should be avoided.

```
// qset.ino - Demonstrate Queue Set

// LED GPIOs:
#define GPIO_LED1     18
#define GPIO_LED2     19
#define GPIO_LED3     21

// Button GPIOs:
#define GPIO_BUT1     27
#define GPIO_BUT2     26
#define GPIO_BUT3     25

#define N_BUTTONS     3
#define Q_DEPTH       8

// ISR Routines, forward decls:
static void IRAM_ATTR isr_gpio1();
static void IRAM_ATTR isr_gpio2();
static void IRAM_ATTR isr_gpio3();

typedef void (*isr_t)();    // ISR routine type

static struct s_button {
  int           butn_gpio;  // Button
  int           led_gpio;   // LED
  QueueHandle_t qh;         // Queue
  isr_t         isr;        // ISR routine
} buttons[N_BUTTONS] = {
  { GPIO_BUT1, GPIO_LED1, nullptr, isr_gpio1 },
  { GPIO_BUT2, GPIO_LED2, nullptr, isr_gpio2 },
  { GPIO_BUT3, GPIO_LED3, nullptr, isr_gpio3 }
};

// Event Task:

```

```
static void evtask(void *arg) {
  QueueSetHandle_t hqset = (QueueSetHandle_t*)arg;
  QueueSetMemberHandle_t mh;
  bool bstate;
  BaseType_t rc;

  for (;;) {
    // Wait for an event from our 3 queues:
    mh = xQueueSelectFromSet(hqset,portMAX_DELAY);
    for ( unsigned ux=0; ux<N_BUTTONS; ++ux ) {
      s_button& button = buttons[ux];

      if ( mh == button.qh ) {
        rc = xQueueReceive(mh,&bstate,0);
        assert(rc == pdPASS);
        digitalWrite(button.led_gpio,bstate);
        break;
      }
    }
  }
}

// Generalized ISR for each GPIO

inline static BaseType_t IRAM_ATTR isr_gpiox(uint8_t gpiox) {
  s_button& button = buttons[gpiox];
  bool state = digitalRead(button.butn_gpio);
  BaseType_t woken = pdFALSE;

  (void)xQueueSendToBackFromISR(button.qh,&state,&woken);
  return woken;
}

// ISR specific to Button 1

static void IRAM_ATTR isr_gpio1() {
  if ( isr_gpiox(0) )
    portYIELD_FROM_ISR();
}

// ISR specific to Button 2

static void IRAM_ATTR isr_gpio2() {
  if ( isr_gpiox(1) )
    portYIELD_FROM_ISR();
}
```

```

// ISR specific to Button 3

static void IRAM_ATTR isr_gpio3() {
  if ( isr_gpiox(2) )
    portYIELD_FROM_ISR();
}

// Program Initialization

void setup() {
  int app_cpu = xPortGetCoreID();
  QueueSetHandle_t hqset;
  BaseType_t rc;

  hqset = xQueueCreateSet(Q_DEPTH*N_BUTTONS);
  assert(hqset);

  // For each button + LED pair:
  for ( unsigned ux=0; ux<N_BUTTONS; ++ux ) {
    s_button& button = buttons[ux];

    button.qh = xQueueCreate(Q_DEPTH,sizeof(bool));
    assert(button.qh);
    rc = xQueueAddToSet(button.qh,hqset);
    assert(rc == pdPASS);
    pinMode(button.led_gpio,OUTPUT);
    digitalWrite(button.led_gpio,1);
    pinMode(button.butn_gpio,INPUT_PULLUP);
    attachInterrupt(button.butn_gpio,button.isr,CHANGE);
  }

  // Start the event task
  rc = xTaskCreatePinnedToCore(
    evtask,     // Function
    "evtask",   // Name
    4096,       // Stack size
    (void*)hqset, // Argument
    1,          // Priority
    nullptr,    // Handle ptr
    app_cpu     // CPU
  );
  assert(rc == pdPASS);
}

// Not used
```

```

void loop() {
  vTaskDelete(nullptr);
}
```

Listing 10-1. Program qset.ino demonstrating the use of FreeRTOS Queue Sets

Summary

Queue sets permit the application designer to specify a set of resources for a task to block its own execution on. The activated resource handle is returned from the call from xQueueSelectFromSet(). When using a mix of queue, semaphore or mutex resources, the caller must determine which API call to make on that member handle. This mode of operation makes it possible to design an efficient event loop, even when multiple resources are involved.

Exercises

1. Can multiple tasks call xQueueSelectFromSet() at the same time?
2. Why were three message queues used instead of one for button press events in the demo program?
3. Does a queue need to be empty to be added to a queue set with xQueueAddToSet()?
4. In the call to xQueueAddToSet(), which is the handle to the queue set? Argument one or two?
5. Both handle arguments to xQueueAddToSet() are valid, yet the call is failing. What are the possible reasons why?
6. What resources can the data type QueueSetMemberHandle_t represent?
7. A queue set must monitor one queue with a depth 3, a binary semaphore, and a counting semaphore with a count of 5. What must the queue set depth be to avoid lost events?
8. Are queue sets recommended for use with mutexes when priority must always be respected?

Chapter 11 • Task Events

Move, you laggard!

Working in projects involving FreeRTOS, you might be left with the impression that tasks could benefit from a simpler event signalling mechanism. Given that the API for direct task event notifications arrived at FreeRTOS version V8.2.0, the need must have been felt by the community and designers alike.[1] The semaphore, mutex, queue, and event group, all involve a separate object that indirectly affects task execution. The task event notification API on the other hand, provides a lighter weight replacement, working directly with tasks. A task or ISR can directly notify another task. There are some restrictions but these tend not to be a problem for most applications.

Task Notification

The ESP32 Arduino framework defines the FreeRTOS macro configUSE_TASK_NOTIFICATIONS=1, causing task notification support to be included. Many designs simply need a task to wait (block its execution) until it is notified, which is traditionally done with a binary or counting semaphore, for example. The task notification API simplifies this by including a built-in 32-bit event notification value for each task, initialized to zero when the task is created.

Because this notification value is built-in in each task, no additional RAM is required. Contrast this with using a binary semaphore where the semaphore object would need to be allocated in addition to the task. The only additional requirement for task notifies is to call the appropriate function for waiting and notifying.

Restrictions

As wonderful as direct task/ISR to task notification is, it cannot be used in all situations. Here are the limitations:

- You cannot send a notification to an ISR (it is not a task). An ISR can notify a task, but not the other way around.
- Only one task may be notified by a notify call (event groups can notify multiple tasks).
- Notification events cannot be buffered like a queue.
- The notifying task cannot block its execution waiting for the receipt of the event by the receiving task.

When any of these issues are involved, you should use the appropriate FreeRTOS object instead of the task notify API.

Waiting

The first step in using task event notification is to set the task to wait for a notification. That function is provided by one of two possible functions:

- ulTaskNotifyTake()
- xTaskNotifyWait()

The first function ulTaskNotifyTake() is the simplest of the two to use.

ulTaskNotifyTake()

A task requiring notification, can block its execution by calling ulTaskNotifyTake(). The call requires two arguments:

```
uint32_t ulTaskNotifyTake(
  BaseType_t xClearCountOnExit, // pdFALSE or pdTRUE
  TickType_t xTicksToWait
);
```

The first argument operates on the 32-bit task notification word (also referred to as the task event word) according to Table 11-1. The second parameter is the familiar timeout value in ticks. In all calls to ulTaskNotifyTake(), the execution of the calling task is blocked while the task event word remains at the value zero. When the value becomes non-zero, the argument xClearCountOnExit determines how the task event word is updated.

xClearCountOnExit Value	Effect on 32-bit task notification word
pdFALSE	--value, blocking execution when the value is zero before the decrement and return.
pdTRUE	Value=0, blocking execution while zero, but clearing the value before return.

Table 11-1. The meanings of ulTaskNotifyTake() argument xClearCountOnExit.

The value returned from the function call is the value of the event notification word before it was cleared or decremented. When a timeout occurs, the returned value will be zero.

Binary Notification

When the argument xClearCountOnExit value is pdTRUE, the call to ulTaskNotifyTake() operates like the binary semaphore take operation. If the task notify event word is already non-zero, the call returns immediately and clears the event word to zero (the non-zero value is returned). If the task notify word was zero at the time of the call, the task execution is blocked until the task is notified (subject to the timeout argument). If a timeout occurs, the return value will be zero (reflecting the value of the event word at the time of return). In this mode of operation, the call operates as a binary semaphore. Even when the task is notified twice before calling ulTaskNotifyTake(), the returned value will be 2, but the task event word is cleared to zero upon return. This effectively provides only one notify event.

Counting Notification

If the requirement is that multiple task notifies should cause multiple task wake-ups, then the xClearCountOnExit value should be provided with the value pdFALSE. In this scenario, the caller blocks while the task event value is zero. Each notify of the task would increment the task event value, but is decremented once upon each return from the waiting task's return.

Give Notify

One option for notifying a task directly is the function xTaskNotifyGive(), which has the following function prototype:

```
        BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
```

This function requires the handle of the task to be notified and always returns pdTRUE (the FreeRTOS manual indicates that xTaskNotifyGive() is defined as a macro).

Note: When using xTaskNotifyGive(), the notified task should be using ulTaskNotifyTake() rather than xTaskNotifyWait().

Demonstration 1

A boiled down and simple demonstration is provided in Listing 11-1, which uses the Serial Monitor. Figure 11-1 has the schematic for wiring the optional LED.

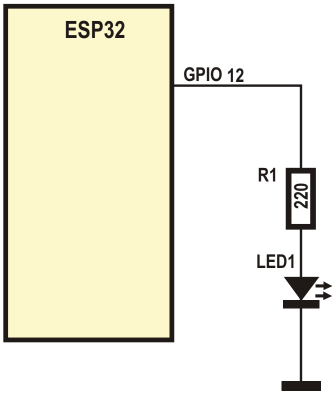


Figure 11-1. Schematic diagram for the tasknfy1.ino and tasknfy2.ino programs.

The program uses the Arduino provided loopTask() notifying task1 from the function loop(). The task1() function blocks until notified and then alternately flashes the LED when notified. The loop() function notifies the task (line 42) using task handle htask1. That handle is created in the setup() function from line 34.

```
0040: void loop() {
0041:   delay(1000);
0042:   xTaskNotifyGive(htask1);
0043: }
```

The task1 code is almost as trivial, operating a forever loop, and blocking at the call to ulTaskNotifyTake() in line 11:

```
0007: static void task1(void *arg) {
0008:   uint32_t rv;
0009:
0010:   for (;;) {
0011:     rv = ulTaskNotifyTake(pdTRUE,portMAX_DELAY);
0012:     digitalWrite(GPIO_LED,digitalRead(GPIO_LED)^HIGH);
0013:     printf("Task notified: rv=%u\n",unsigned(rv));
0014:   }
0015: }
```

The task execution is blocked until the task notify event occurs. The value assigned to rv is the value of the event notification word *prior* to it being cleared (due to the pdTRUE argument). Notice how simple and elegant this is – there are no other semaphores or handles involved.

The serial monitor session should look like this:

```
tasknfy1.ino:
Task notified: rv=1
Task notified: rv=1
...
```

If you were to notify a task from an ISR, you would use xTaskNotifyGiveFromISR() function instead.

```
void vTaskNotifyGiveFromISR(
  TaskHandle_t xTaskToNotify,
  BaseType_t *pxHigherPriorityTaskWoken
);
```

The second argument is the ISR typical wakeup flag, indicating whether or not the scheduler should be invoked.

```
0001: // tasknfy1.ino
0002:
0003: #define GPIO_LED        12
0004:
0005: static TaskHandle_t htask1;
```

```

static void task1(void *arg) {
  uint32_t rv;

  for (;;) {
    rv = ulTaskNotifyTake(pdTRUE,portMAX_DELAY);
    digitalWrite(GPIO_LED,digitalRead(GPIO_LED)^HIGH);
    printf("Task notified: rv=%u\n",unsigned(rv));
  }
}

void setup() {
  int app_cpu = 0;
  BaseType_t rc;

  app_cpu = xPortGetCoreID();
  pinMode(GPIO_LED,OUTPUT);
  digitalWrite(GPIO_LED,LOW);

  delay(2000); // Allow USB to connect
  printf("tasknfy1.ino:\n");

  rc = xTaskCreatePinnedToCore(
    task1,    // Task function
    "task1",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    &htask1,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
}

void loop() {
  delay(1000);
  xTaskNotifyGive(htask1);
}
```

Listing 11-1. The xTaskNotifyGive() and ulTaskNotifyTake() demonstration program tasknfy1.ino.

Demonstration 2

Each call to xTaskNotifyGive(), increments the task event notification word. The next Serial Monitor example will illustrate two things:

- Multiple tasks notifying task1
- How the event notify word can increment to values above 1

Listing 11-2 contains the full listing and uses the same wiring as provided in Figure 11-1. The loop() function remains the same, but we've added a task2, which performs a similar function:

```
0017: static void task2(void *arg) {
0018:   unsigned count = 0;
0019:
0020:   for (;; count += 100) {
0021:     delay(500+count);
0022:     xTaskNotifyGive(htask1);
0023:   }
0024: }
```

The task2() function varies from the loop() function by adding a variable delay (line 21), to enhance the effect of multiple task notifications. When the demonstration is flashed and started, the Serial Monitor should display a session like this:

```
tasknfy2.ino:
Task notified: rv=2
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=2
Task notified: rv=1
Task notified: rv=2
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
...
```

As the task runs, the task notifies should accumulate so that the reported rv value will sometimes be the value 2 instead of 1.

```
// tasknfy2.ino

#define GPIO_LED        12

static TaskHandle_t htask1;

static void task1(void *arg) {
  uint32_t rv;

  for (;;) {
    rv = ulTaskNotifyTake(pdTRUE,portMAX_DELAY);
    digitalWrite(GPIO_LED,digitalRead(GPIO_LED)^HIGH);
    printf("Task notified: rv=%u\n",unsigned(rv));
  }
}

static void task2(void *arg) {
  unsigned count = 0;

  for (;; count += 100) {
    delay(500+count);
    xTaskNotifyGive(htask1);
  }
}

void setup() {
  int app_cpu = 0;
  BaseType_t rc;

  app_cpu = xPortGetCoreID();
  pinMode(GPIO_LED,OUTPUT);
  digitalWrite(GPIO_LED,LOW);

  delay(2000); // Allow USB to connect
  printf("tasknfy2.ino:\n");

  rc = xTaskCreatePinnedToCore(
    task1,    // Task function
    "task1",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    &htask1,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);

  rc = xTaskCreatePinnedToCore(
    task2,    // Task function
    "task2",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    nullptr,  // no handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
}

void loop() {
  delay(500);
  xTaskNotifyGive(htask1);
}
```

Listing 11-2. The demonstration of multiple tasks notifying in program tasknfy2.ino.

Demonstration 3

The task notify event word can be used as a counting semaphore. Listing 11-3 is the same as Listing 11-2, except for the following call performed in task1(), where argument one is provided with pdFALSE:

```
    rv = ulTaskNotifyTake(pdFALSE,portMAX_DELAY);
```

When pdFALSE is used as the first argument value, the event word is simply decremented. It may be somewhat difficult to notice, but when the program runs, you will see that lines where rv=2 are reported, one more immediately follows with rv=1, before the task pauses again. This happens because the task is pending an additional notify until the event word becomes zero.

```
tasknfy3.ino:
Task notified: rv=2
Task notified: rv=1 // immediately after previous
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=1
Task notified: rv=2
Task notified: rv=1 // immediately after previous
Task notified: rv=1
...
```

```
// tasknfy3.ino

#define GPIO_LED        12

static TaskHandle_t htask1;

static void task1(void *arg) {
  uint32_t rv;

  for (;;) {
    rv = ulTaskNotifyTake(pdFALSE,portMAX_DELAY);
    digitalWrite(GPIO_LED,digitalRead(GPIO_LED)^HIGH);
    printf("Task notified: rv=%u\n",unsigned(rv));
  }
}

static void task2(void *arg) {
  unsigned count = 0;

  for (;; count += 100) {
    delay(500+count);
    xTaskNotifyGive(htask1);
  }
}

void setup() {
  int app_cpu = 0;
  BaseType_t rc;

  app_cpu = xPortGetCoreID();
  pinMode(GPIO_LED,OUTPUT);
  digitalWrite(GPIO_LED,LOW);

  delay(2000); // Allow USB to connect
  printf("tasknfy3.ino:\n");

  rc = xTaskCreatePinnedToCore(
    task1,    // Task function
    "task1",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    &htask1,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);

  rc = xTaskCreatePinnedToCore(
    task2,    // Task function
    "task2",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    nullptr,  // no handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
}

void loop() {
  delay(500);
  xTaskNotifyGive(htask1);
}
```

Listing 11-3. Using the task notify event word as a counting semaphore in tasknfy3.ino.

Going Beyond Simple Notify
What has been presented so far works well for simple task wake-up events but some applications have more sophisticated requirements. The xTaskNotifyWait() function is more complicated to use but offers additional flexibility. The function prototype is as follows:

```
BaseType_t xTaskNotifyWait(
  uint32_t ulBitsToClearOnEntry,
  uint32_t ulBitsToClearOnExit,
  uint32_t *pulNotificationValue,
  TickType_t xTicksToWait
);
```

The return value differs from the ulTaskNotifyTake() function, returning success or fail instead:

- pdTRUE – a notification was received (or was already present)
- pdFALSE – a timeout occurred

The third argument is a pointer to a uint32_t, that will receive the event word. It can be supplied as nullptr (or NULL) when you don't require it.

Argument 3 – ulBitsToClearOnExit
This argument defines the bits to be cleared from the event word, upon receiving an event (no bits are cleared for timeouts). The value returned by pulNotificationValue (argument 3) will include the value of those bits, *prior* to them being cleared. This allows your code to determine which event flags were set. Specific bits can be set in the event word using the xTaskNotify() function, to be discussed next.

Argument 2 – ulBitsToClearOnEntry
This argument is often supplied as zero, depending upon requirements. This value specifies which bits should be cleared in the event word, prior to the actual waiting of an event. This provides a means to clear bits that might be pending that should be ignored and force a wait.

Smart Notify
To provide the FreeRTOS user with the ability to set individual event bits, the xTaskNotify() function should be used:

```
BaseType_t xTaskNotify(
  TaskHandle_t xTaskToNotify,
  uint32_t ulValue,
  eNotifyAction eAction
);
```

The function returns two possible values:

- pdPASS – operation succeeded.
- pdFAIL – operation failed.

The task handle is provided in argument one, as before.

Argument eAction
Argument two provides the event value, while argument eAction describes how that value will be applied. The ways that ulValue are applied are described in Table 112.

eAction	Description
eNoAction	The task is notified but the event notification word is not used (ulValue is not used).
eSetBits	The task's event word is updated with the bitwise OR of ulValue.
eIncrement	The task event word is incremented by one (ulValue is not used)
eSetValueWithOverwrite	The task's event word is overwritten with ulValue, even if the task already had a notification pending.
eSetValueWithoutOverwrite	If the task already has a pending notification then its value is not changed and the call returns pdFAIL.

Table 11-2. The eAction values and their operations.

Demonstration 4

Listing 11-4 is provided for tasknfy4.ino, which continues to use the same circuit. This demonstration differs from the previous ones in that it sets a specific bit within the task event word, indicating the source of the event. Let's examine the task1() code first:

```
0007: static void task1(void *arg) {
0008:   uint32_t rv;
0009:   BaseType_t rc;
0010:
0011:   for (;;) {
0012:     rc = xTaskNotifyWait(0,0b0011,&rv,portMAX_DELAY);
0013:     digitalWrite(GPIO_LED,digitalRead(GPIO_LED)^HIGH);
0014:     printf("Task notified: rv=%u\n",unsigned(rv));
0015:     if ( rv & 0b0001 )
0016:       printf("  loop() notified this task.\n");
0017:     if ( rv & 0b0010 )
0018:       printf("  task2() notified this task.\n");
0019:   }
0020: }
```

In this code, we wait for a bit 0 or bit 1 event in line 12 (argument 0b0011). The value of the event word is returned into rv, which is then tested in line 15 and 17. Based upon the bit(s) found set, we can determine the event source and report them in lines 16 and 18. The loop() function posts the notify event with this code:

```
0071:   rc = xTaskNotify(htask1,0b0001,eSetBits);
```

This simply ORs in a 1-bit into the task event word's bit-0. Likewise, in task2() a similar operation occurs, except that it ORs in a 1-bit in bit-1 (value 0b0010) of the task event word:

```
0028:     rc = xTaskNotify(htask1,0b0010,eSetBits);
```

When this demonstration runs, you should see a session similar to this:

```
tasknfy4.ino:
Task notified: rv=3
  loop() notified this task.
  task2() notified this task.
Task notified: rv=1
  loop() notified this task.
Task notified: rv=2
  task2() notified this task.
Task notified: rv=1
  loop() notified this task.
Task notified: rv=2
  task2() notified this task.
```

```
Task notified: rv=1
  loop() notified this task.
Task notified: rv=1
  loop() notified this task.
Task notified: rv=2
  task2() notified this task.
Task notified: rv=1
  loop() notified this task.
Task notified: rv=3
  loop() notified this task.
  task2() notified this task.
...
```

Setting bits in this way can be extremely useful to device drivers. A UART driver might report a different flag for received data, transmitter buffer empty, frame, or parity error or break. To perform the same notification from an ISR requires a FromISR suffix on the function name:

```
BaseType_t xTaskNotifyFromISR(
  TaskHandle_t xTaskToNotify,
  uint32_t ulValue,
  eNotifyAction eAction,
  BaseType_t * pxHigherPriorityTaskWoken
);
```

Apart from the pxHigherPriorityTaskWoken argument, the usage is the same.

```
// tasknfy4.ino

#define GPIO_LED        12

static TaskHandle_t htask1;

static void task1(void *arg) {
  uint32_t rv;
  BaseType_t rc;

  for (;;) {
    rc = xTaskNotifyWait(0,0b0011,&rv,portMAX_DELAY);
    digitalWrite(GPIO_LED,digitalRead(GPIO_LED)^HIGH);
    printf("Task notified: rv=%u\n",unsigned(rv));
    if ( rv & 0b0001 )
      printf("  loop() notified this task.\n");
    if ( rv & 0b0010 )
      printf("  task2() notified this task.\n");
  }
}

static void task2(void *arg) {
  unsigned count;
  BaseType_t rc;

  for (;; count += 100u ) {
    delay(500+count);
    rc = xTaskNotify(htask1,0b0010,eSetBits);
    assert(rc == pdPASS);
  }
}

void setup() {
  int app_cpu = 0;
  BaseType_t rc;

  app_cpu = xPortGetCoreID();
  pinMode(GPIO_LED,OUTPUT);
  digitalWrite(GPIO_LED,LOW);

  delay(2000); // Allow USB to connect
  printf("tasknfy4.ino:\n");

  rc = xTaskCreatePinnedToCore(
    task1,    // Task function
    "task1",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    &htask1,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);

  rc = xTaskCreatePinnedToCore(
    task2,    // Task function
    "task2",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    nullptr,  // no handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
}
```

```
0066:
0067: void loop() {
0068:   BaseType_t rc;
0069:
0070:   delay(500);
0071:   rc = xTaskNotify(htask1,0b0001,eSetBits);
0072:   assert(rc == pdPASS);
0073: }
```

Listing 11-4. Task events using eSetBits in tasknfy4.ino.

Demonstration 5

Demonstration 5 uses interrupts to provide a final example of what we've been discussing in this chapter. ISR routines are frequently the source of events but these must be communicated to a task with the minimum of code. This program uses the task event word bits to indicate when one of three buttons change state, rather than a queue. Once the task is notified, it reads the state of the push button input GPIO.

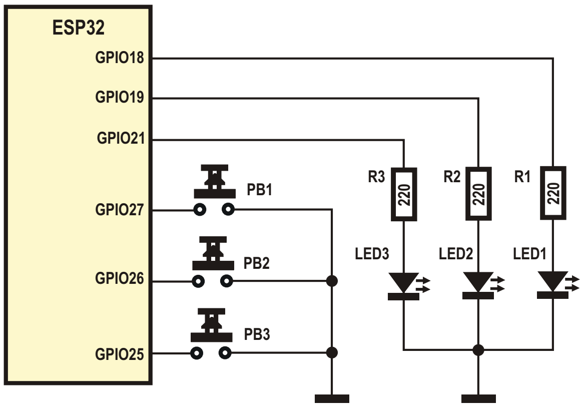


Figure 11-2. The circuit for the tasknfy.ino program.

The ISR notifies the task by means of xTaskNotifyFromISR(), using a 1-bit shifted up by the button index (line 60).

```
0056: inline static BaseType_t IRAM_ATTR isr_gpiox(uint8_t gpiox) {
0057:   s_button& button = buttons[gpiox];
0058:   BaseType_t woken = pdFALSE;
0059:
0060:   xTaskNotifyFromISR(htask1,1 << button.buttonx,eSetBits,&woken);
0061:   return woken;
0062: }
```

The task receiving the events is very similar to before, blocking on the call to xTaskNotify-Wait() at line 42. There it blocks for event bits 2, 1 or 0, with the events returned via rv. A simple loop then tests for each of those bits in lines 44 to 50, reporting the current state of the button.

```
0037: static void task1(void *arg) {
0038:   uint32_t rv;
0039:   BaseType_t rc;
0040:
0041:   for (;;) {
0042:     rc = xTaskNotifyWait(0,0b0111,&rv,portMAX_DELAY);
0043:     printf("Task notified: rv=%u\n",unsigned(rv));
0044:     for ( unsigned x=0; x<3; ++x ) {
0045:       if ( rv & (1 << x) )
0046:         printf("  Button %u notified, reads %d\n",
0047:           x,digitalRead(buttons[x].butn_gpio));
0048:         digitalWrite(buttons[x].led_gpio,
0049:           digitalRead(buttons[x].butn_gpio));
0050:     }
0051:   }
0052: }
```

The demonstration requires the use of the Serial Monitor in addition to the LED indicators. A photo of the breadboard setup appears in Figure 11-3. The Serial Monitor should report a session output similar to the following, as you push buttons:

```
tasknfy5.ino:
Task notified: rv=4
  Button 2 notified, reads 0
Task notified: rv=4
  Button 2 notified, reads 1
Task notified: rv=2
  Button 1 notified, reads 0
Task notified: rv=2
  Button 1 notified, reads 1
Task notified: rv=1
  Button 0 notified, reads 0
Task notified: rv=1
  Button 0 notified, reads 1
Task notified: rv=4
  Button 2 notified, reads 0
Task notified: rv=1
  Button 0 notified, reads 0
Task notified: rv=4
  Button 2 notified, reads 1
...
```

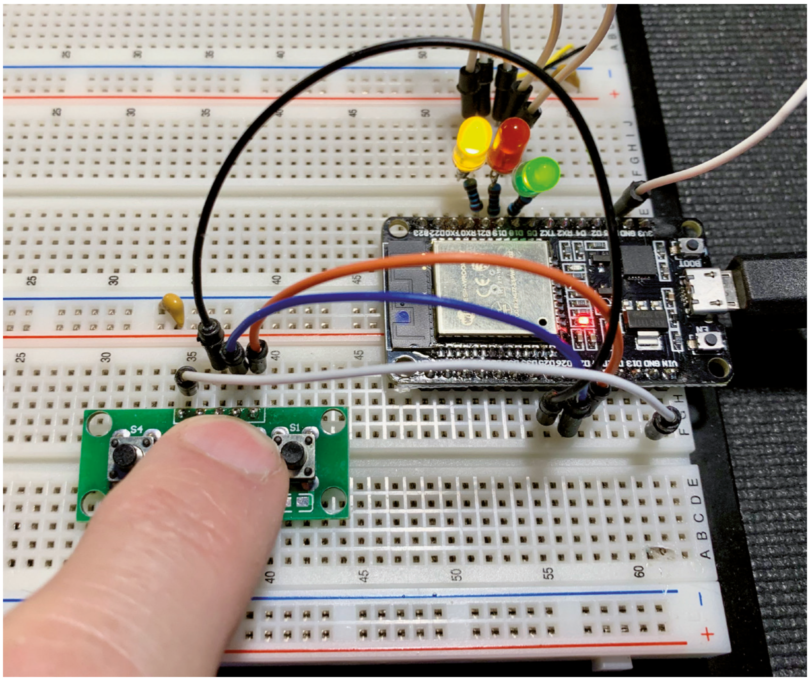


Figure 11-3. Photo of breadboarded tasknfy5.ino setup.

```
 // tasknfy5.ino

// LED GPIOs:
#define GPIO_LED1     18
#define GPIO_LED2     19
#define GPIO_LED3     21

// Button GPIOs:
#define GPIO_BUT1     27
#define GPIO_BUT2     26
#define GPIO_BUT3     25

#define N_BUTTONS     3

// ISR Routines, forward decls:
static void IRAM_ATTR isr_gpio1();
static void IRAM_ATTR isr_gpio2();
static void IRAM_ATTR isr_gpio3();

typedef void (*isr_t)();    // ISR routine type

static struct s_button {
  int           butn_gpio;  // Button
  int           led_gpio;   // LED
  unsigned      buttonx;    // Button index
  isr_t         isr;        // ISR routine
} buttons[N_BUTTONS] = {
  { GPIO_BUT1, GPIO_LED1, 0u, isr_gpio1 },
  { GPIO_BUT2, GPIO_LED2, 1u, isr_gpio2 },
  { GPIO_BUT3, GPIO_LED3, 2u, isr_gpio3 }
};

static TaskHandle_t htask1;

// Task1 for sensing buttons

static void task1(void *arg) {
  uint32_t rv;
  BaseType_t rc;

  for (;;) {
    rc = xTaskNotifyWait(0,0b0111,&rv,portMAX_DELAY);
    printf("Task notified: rv=%u\n",unsigned(rv));
    for ( unsigned x=0; x<3; ++x ) {
      if ( rv & (1 << x) )
        printf("  Button %u notified, reads %d\n",
          x,digitalRead(buttons[x].butn_gpio));
        digitalWrite(buttons[x].led_gpio,
          digitalRead(buttons[x].butn_gpio));
    }
  }
}

// Generalized ISR for each GPIO

inline static BaseType_t IRAM_ATTR isr_gpiox(uint8_t gpiox) {
  s_button& button = buttons[gpiox];
  BaseType_t woken = pdFALSE;

  xTaskNotify(htask1,1 << button.buttonx,eSetBits);
  return woken;
}

// ISR specific to Button 1

static void IRAM_ATTR isr_gpio1() {
  if ( isr_gpiox(0) )
    portYIELD_FROM_ISR();
}
```

```

// ISR specific to Button 2

static void IRAM_ATTR isr_gpio2() {
  if ( isr_gpiox(1) )
    portYIELD_FROM_ISR();
}

// ISR specific to Button 3

static void IRAM_ATTR isr_gpio3() {
  if ( isr_gpiox(2) )
    portYIELD_FROM_ISR();
}

// Initialization

void setup() {
  int app_cpu = xPortGetCoreID();
  BaseType_t rc;

  // For each button + LED pair:
  for ( unsigned ux=0; ux<N_BUTTONS; ++ux ) {
    s_button& button = buttons[ux];

    pinMode(button.led_gpio,OUTPUT);
    digitalWrite(button.led_gpio,1);
    pinMode(button.butn_gpio,INPUT_PULLUP);
    attachInterrupt(button.butn_gpio,button.isr,CHANGE);
  }

  delay(2000); // Allow USB to connect
  printf("tasknfy5.ino:\n");

  rc = xTaskCreatePinnedToCore(
    task1,    // Task function
    "task1",  // Name
    3000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    &htask1,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
}

```

```
// Not used:

void loop() {
  vTaskDelete(nullptr);
}
```

Listing 11.5. The Interrupt Driven Button Processor tasknfy5.ino.

Summary

The task notify facility of FreeRTOS provides a convenient and welcome addition to the mature FreeRTOS API. The fact that it is baked to the tasking framework makes it compact, while the bitwise operations make it possible to post individual events in a very efficient manner.

Exercises

1. Where is the storage allocated for task notify events?
2. How does the calling of xTaskNotifyGive() affect the receiving task's event word?
3. How many bits are available in the task event word?
4. Why would you ever use a non-zero ulBitsToClearOnEntry value when calling xTaskNotifyWait()?
5. Why should ulTaskNotifyTake() be used in preference to xTaskNotifyWait()?
6. What happens when eNoAction is used to notify a task?
7. What does xTaskNotifyWait() return when it times out?
8. What does xTaskNotify() return when it fails?
9. Why is the use of ulTaskNotifyTake() or xTaskNotifyWait() efficient use of CPU time?

Web Resources

[1] https://www.freertos.org/RTOS-task-notifications.html

Chapter 12 • Event Groups

Please share with the group, 13, about today's events.

Event groups were introduced in Feb 2014, in FreeRTOS V8.0.0 before task notifications became available. While task notification remains the lighter weight API, it doesn't permit the notifying of multiple tasks with one event. The event group, on the other hand, can perform group notifications as well as specialized bit combinations of events. This chapter will examine event groups along with two practical applications of it.

EventBits_t Type

The event group is implemented as a 32-bit word, but only bits 0 through to 23 are available as event flags to the end-user. Each flag bit is a 1-bit if the event has occurred, or zero if it has not (or has been cleared). This permits the manipulation of up to 24 events in one event group object.

Creating an Event Group Object

The creation of an event group object is a simple matter of making a call to obtain the created event group's handle:

```
EventGroupHandle_t xEventGroupCreate(void);
```

Once you have an event group handle, then the following two actions are normally used:

- Notifying an event group – xEventGroupSetBits()
  (or xEventGroupSetBitsFromISR())
- Waiting for an event – xEventGroupWaitBits() (*not available from an ISR).*

Static creation is not supported by the ESP32 Arduino environment, but you may use it in other FreeRTOS environments, like the ESP-IDF:

```
StaticEventGroup_t event_group;
EventGroupHandle_t h;

h = xEventGroupCreateStatic(&event_group);
```

Notifying an Event Group
Notifying an event group is simply the turning on of certain bits (between 0 and 23) using the function xEventGroupSetBits(). This differs from task notification because this action can potentially notify multiple tasks:

```
EventBits_t xEventGroupSetBits(
  EventGroupHandle_t xEventGroup,  // Event group handle
  const EventBits_t uxBitsToSet    // Bits to be set
);
```

The value uxBitsToSet are ORed with the existing event bits in the word, in the event group object. Once bits are added, tasks waiting on certain bits may be notified.

Waiting for Event Groups
A task wanting to be notified by an event group, will call upon xEventGroupWaitBits():

```
EventBits_t xEventGroupWaitBits(
  const EventGroupHandle_t xEventGroup,  // Handle
  const EventBits_t uxBitsToWaitFor,     // bits to wait for (non-zero only)
  const BaseType_t xClearOnExit,         // pdTRUE / pdFALSE
  const BaseType_t xWaitForAllBits,      // pdTRUE / pdFALSE
  TickType_t xTicksToWait                // timeout
);
```

You are already familiar with the handle and the timeout parameters from the other FreeRTOS calls described in earlier chapters. When argument four (xWaitForAllBits) is set to pdFALSE, then the call simply waits for *any* of the bits provided in uxBitsToWaitFor. The bits being waited for should only include bits from bit-0 to bit-23 inclusive. The value of this argument must not be zero.

When the argument xClearOnExit is pdTRUE, then the bits that were being waited for, are cleared before returning to the caller. When the argument xWaitForAllBits is pdTRUE, then all bits to be waited for in uxBitsToWaitFor must be 1bits, before the event is returned.

The function return value is the EventBits_t value at the time of the timeout or when the uxBitsToWait condition was met. When a timeout is possible, you must check the return value to see if your conditions were met. Or specify portMAX_DELAY for the timeout argument for no timeout.

Table 12-1 lists some examples of argument values that can be supplied and their effect upon the xEventGroupWaitBits() function call. Example 1 (column Ref) specifies pdFALSE for xClearOnExit, so the result of the event group is left unchanged when the bit-1 (0b0010) is sensed true. Example 2 specifies xClearOnExit to pdTRUE, so the event word (column EventBits_t Result) has bit-1 cleared after being triggered (0b1000). Examples 3 and 4 have the argument xWaitForAllBits set to pdTRUE, so that all bits in uxBitsToWaitFor must be set before returning the event. This is not met in example 3, so the call does not return (or times out). Example 4 does return an event because the event word matched with value 0b1011 in bits 1 and 0. Those bits are cleared upon return, leaving the event word with value 0b1000.

Ref	uxBitsToWaitFor	xClearOnExit	xWaitForAllBits	EventBits_t Value	Action	EventBits_t Result
1	0b0011	pdFALSE	pdFALSE	0b1010	Triggered	0b1010
2	0b0011	pdTRUE	pdFALSE	0b1010	Triggered	0b1000
3	0b0011	pdTRUE	pdTRUE	0b1010	Not triggered	0b1010
4	0b0011	pdTRUE	pdTRUE	0b1011	Triggered	0b1000

Table 12-1. Some event examples for xEventGroupWaitBits().

The EventBits_t value returned from the function call reflect the bits *prior* to any bits being cleared. This allows the caller to test what event bits were found set. This is also true for timeouts, but in that case, there would be no bits cleared.

Demonstration 1

To put things into concrete terms, the program in listing 12-1 is provided named eventgr.ino.[1] It uses three LEDs attached as shown in Figure 12-1. This program uses both the Serial Monitor and your WiFi router. You must provide your WiFi credentials by editing the following two lines of the source file:

```
0004: #define WIFI_SSID      "MySSID"
0005: #define WIFI_PASSWD    "MyPassword"
```

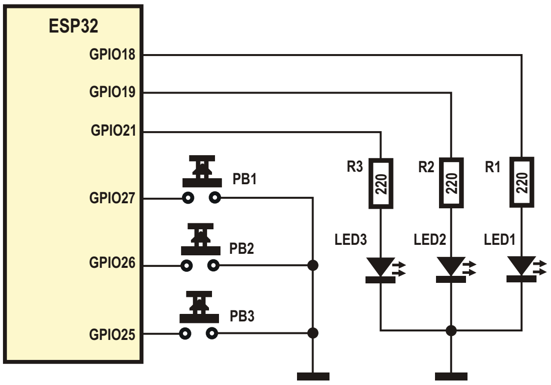


Figure 12-1. The schematic diagram for the eventgr.ino demonstration program.

Save your changes and flash that to the ESP32 (almost any dual-core ESP32 can be used). The demonstration program will perform the following general steps:

1. Initialize three LED output GPIOs.
2. Create an event group.
3. Wait briefly for the USB to serial interface to connect.

4. Create the http_server() task.
5. Create the udp_broadcast() task.
6. Initialize the connection to your WiFi router.
7. And when connected to the WiFi router, the event group will post the event WIFI_RDY (0b0001).
8. Once that event is posted, both tasks will begin in their respective server functions.

With this procedure, the event group functions as a barrier – preventing the server tasks from trying to serve before the WiFi facility is available. The barrier is released in the function init_http(), line 227:

```
0227:   xEventGroupSetBits(hevt,WIFI_RDY);
```

With the macro expansion, this becomes:

```
0227:   xEventGroupSetBits(hevt,0b0001);
```

By setting bit-0 to 1 in the event group it signals that the WiFi facility is ready. This is only half of the mechanism since we need to see how the notification is received by the two notified tasks. The start of the http_task() has this call:

```
0054:   xEventGroupWaitBits(
0055:     hevt,           // Event group
0056:     WIFI_RDY,       // bits to wait for
0057:     pdFALSE,        // no clear
0058:     pdFALSE,        // wait for all bits
0059:     portMAX_DELAY); // timeout
```

The start of the udp_broadcast() task uses the same call in lines 156 to 161. Let's break this call down:

The call waits for any of the bits in WIFI_RDY (0b0001) to become true (there is only one bit to wait for in this case).

1. No bits are cleared upon return (line 57).
2. Only one bit needs to match (line 58) (although we're only waiting for one bit).
3. And the call will block forever (line 59) until the WiFi ready condition has been met.

Once bit-0 is set to 1, both blocked tasks will return and be allowed to continue. In this manner, all WiFi activity is blocked until it has notified that the WiFi has connected. Note that this code is simple-minded because it does not handle the case where the WiFi might disconnect. For this demonstration, simply reset the ESP32 if this happens.

Once the program starts and is WiFi ready, the program will report it's IP number to the serial monitor. Here it is reported that the HTTP server is available as 192.168.1.21 port 80. The UDP broadcast server also determines its broadcast address as 192.168.1.255 in the example session below:

```
eventgr.ino:
WiFi connecting to SSID: WhereEaglesDare
....
WiFi connected as 192.168.1.21
Server ready: 192.168.1.21 port 80
UDP ready: netmask 255.255.255.0 broadcast 192.168.1.255
```

Once you see that in the session, connect your browser to "http://192.168.1.21/" (port 80 is assumed). Substitute for 192.168.1.21 according to what your serial monitor has reported. The browser should then display a page like the one shown in Figure 12-2.

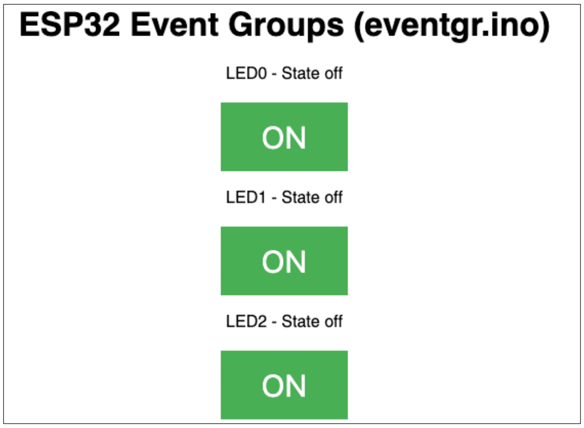


Figure 12-2. The page displayed on the browser.

The page displays the state of the three LEDs and provides buttons for changing their state. To turn on LED0, press the top button. Once that LED is on, the button text will change to "OFF". This is serviced by the task in lines 52 to 152. In addition to parsing the GET request and driving the LEDs, the following code performs one other function:

```
0093:          if ( cp[0] >= '0' && cp[0] <= ('0'+N_LED) ) {
0094:            int ledx = uint8_t(cp[0]) - uint8_t('0');
0095:
0096:            if ( cp[1] == '/'
0097:              && ( cp[2] == '0' || cp[2] == '1' ) ) {
0098:                bool onoff = !!(cp[2] & 1);
0099:              printf("LED%d = %d\n",ledx,onoff);
0100:              if ( onoff != !!digitalRead(leds[ledx]) ) {
0101:                digitalWrite(leds[ledx],onoff?HIGH:LOW);
0102:                changedf = true;
0103:               }
0104:             }
0105:           }
0106:           if ( changedf )
0107:             xEventGroupSetBits(hevt,LED_CHG);
```

If any LED has changed in value (line 100 tests for this), then the bool flag changedf is set to true (line 102). Line 107 sends another event LED_CHG to the group, if a change in the LED states were detected. The value of macro LED_CHG is 0b0010 (line 17).

Within the udp_broadcast() task, the following code is of interest:

```
0184:   for (;;) {
0185:     xEventGroupWaitBits(
0186:       hevt,           // handle
0187:       LED_CHG,        // bits to wait for
0188:       pdTRUE,         // clear bits
0189:       pdFALSE,        // wait for all bits
0190:       portMAX_DELAY); // timeout
0191:     char temp[16];
0192:
0193:     // Send UDP packet:
0194:     udp.beginPacket(broadcast,9000);
0195:     for ( short x=0; x<N_LED; ++x ) {
0196:       bool state = !!digitalRead(leds[x]);
0197:       snprintf(temp,sizeof temp,
0198:         "LED%d=%d\n",
0199:         x,state);
0200:       udp.write((uint8_t const*)temp,strlen(temp));
0201:     }
0202:     udp.write((uint8_t const*)"--\n",3);
0203:     udp.endPacket();
0204:   }
```

The for loop blocks at line 185-190 until a LED_CHG event is sensed. Note that in this call, the clear bits argument is passed as pdTRUE (line 188), so that the event is cleared after being returned. This way, the loop can know that the LED configuration has changed and generate a UDP packet to notify of this.

To view the UDP packets, a good tool to use is the netcat command (usually installed as nc). You may need to install this if you don't already have it:

- On MacOS it may be installed as /usr/bin/nc. Otherwise, you may install it from HomeBrew (brew install netcat).
- On Linux: sudo apt install netcat (this will depend on your Linux distribution used).

- Windows users may need to do one of the following:
    - install WSL + sudo apt install netcat
    - Use the Windows netcat.exe program (may require whitelisting with your antivirus program)
    - Or download an open/shareware version of netcat.

Before flashing and starting the demonstration, perform the following in a terminal session – using the broadcast address reported by the serial monitor do:

```
$ nc -u -l 192.168.1.255 9000
Ready:
```

The -u option tells netcat (nc) to be expecting UDP packets (the defaults is TCP/IP). The -l (el) option tells the command to *listen* for UDP packets rather than sending them. The address 192.168.1.255 is the broadcast address computed for your WiFi router, and the demonstration program broadcasts the packets to port 9000 (set by line 180 of the program).

Once the demonstration has connected to the WiFi router, the first packet is a message "Ready:" (as shown in the example session above). Clicking on the browser page's buttons will cause updates to the LEDs. Clicking on LED1 (middle button) produces the following result (it shows that LED1=1).

```
$ nc -u -l 192.168.1.255 9000
Ready:
LED0=0
LED1=1
LED2=0
--
```

Demo Conclusion
This demonstration has used the event group in two different ways:

1. As a barrier (WiFi ready indication).
2. As an event notification (LED change).

The event group permits sophisticated ways to efficiently notify one or more tasks. In the next section, the synchronize capability of event groups will be demonstrated.

```
0001: // eventgr.ino
0002:
0003: // Update with your WIFI credentials
0004: #define WIFI_SSID     "MySSID"
0005: #define WIFI_PASSWD   "MyPassword"
0006:
0007: // LED GPIOs:
```

```
#define GPIO_LED1     18
#define GPIO_LED2     19
#define GPIO_LED3     21

#define N_LED         3

#include <WiFi.h>

#define WIFI_RDY      0b0001
#define LED_CHG       0b0010

static EventGroupHandle_t hevt;
static WiFiServer http(80);
static WiFiUDP udp;

static int leds[N_LED] =
  { GPIO_LED1, GPIO_LED2, GPIO_LED3 };

static char const
  *ssid = WIFI_SSID,
  *passwd = WIFI_PASSWD;

static bool getline(String& s,WiFiClient client) {
  char ch;
  bool flag = false;

  s.clear();
  while ( client.connected() ) {
    if ( client.available() ) {
      ch = client.read();
      flag = true;

      if ( ch == ‘\r’ )
        continue;         // Ignore CR
      if ( ch == ‘\n’ )
        break;
      s += ch;
    } else {
      taskYIELD();
    }
  }
  return client.connected() && flag;
}

static void http_server(void *arg) {

```

```
xEventGroupWaitBits(
  hevt,           // Event group
  WIFI_RDY,       // bits to wait for
  pdFALSE,        // no clear
  pdFALSE,        // wait for all bits
  portMAX_DELAY); // timeout

auto subnet = WiFi.subnetMask();
printf("Server ready: %s port 80\n",
  WiFi.localIP().toString().c_str());

for (;;) {
  WiFiClient client = http.available();

  if ( client ) {
    // A client has connected:
    String line, header;
    bool gothdrf = false;

    printf("New client connect from %s\n",
      client.remoteIP().toString().c_str());

    while ( client.connected() ) {
      if ( getline(header,client) ) {
        while ( getline(line,client) && line.length() > 0 )
          ;
      }
      if ( !client.connected() )
        break;

      client.println("HTTP/1.1 200 OK");
      client.println("Content-type:text/html");
      client.println("Connection: close");
      client.println();

      if ( !strncmp(header.c_str(),"GET /led",8) ) {
        const char *cp = header.c_str() + 8;
        bool changedf = false;

        if ( cp[0] >= ‘0’ && cp[0] <= (‘0’+N_LED) ) {
          int ledx = uint8_t(cp[0]) - uint8_t(‘0’);

          if ( cp[1] == ‘/’
            && ( cp[2] == ‘0’ || cp[2] == ‘1’ ) ) {
              bool onoff = !!(cp[2] & 1);
            printf("LED%d = %d\n",ledx,onoff);
            if ( onoff != !!digitalRead(leds[ledx]) ) {
              digitalWrite(leds[ledx],onoff?HIGH:LOW);
              changedf = true;
            }
          }
        }
        if ( changedf )
          xEventGroupSetBits(hevt,LED_CHG);
      }

      client.println("<!DOCTYPE html><html>");
      client.println("<head><meta name=\"viewport\" "
        "content=\"width=device-width, initial-scale=1\">");
      client.println("<link rel=\"icon\" href=\"data:,\">");
      client.println("<style>html { font-family: Helvetica; "
        "display: inline-block; margin: 0px auto; "
        "text-align: center;}");
      client.println(".button { background-color: "
        "#4CAF50; border: none; color: white; "
        "padding: 16px 40px;");
      client.println("text-decoration: none; "
        "font-size: 30px; margin: 2px; cursor: pointer;}");
      client.println(".button2 {background-color: #555555;}"
        "</style></head>");
      client.println("<body><h1>ESP32 Event Groups "
        "(eventgr.ino)</h1>");

      for ( int x=0; x<N_LED; ++x ) {
        bool state = !!digitalRead(leds[x]);
        char temp[32];

        snprintf(temp,sizeof temp,"<p>LED%d - State ",x);
        client.println(temp);
        client.println(String(state ? "on" : "off") + "</p>");
        client.println("<p><a href=\"");
        snprintf(temp,sizeof temp,"/led%d/%d",x,!state);
        client.println(temp);
        client.println("\"><button class=\"button\">");
        client.println(state?"OFF":"ON");
        client.println("</button></a></p>");
      }

      client.println("</body></html>");
      client.println();
          break;
    }
```

```
      client.stop();
      header = "";
      Serial.println("Client disconnected.");
      Serial.println("");
    }
  }
}

static void udp_broadcast(void *arg) {

  xEventGroupWaitBits(
    hevt,           // Event Group
    WIFI_RDY,       // bits to wait for
    pdFALSE,        // no clear
    pdFALSE,        // wait for all bits
    portMAX_DELAY); // timeout

  // Determine IPv4 broadcast address:

  auto localip = WiFi.localIP();
  auto subnet = WiFi.subnetMask();
  auto broadcast = localip;

  for ( short x=0; x<4; ++x ) {
    broadcast[x] = 0xFF & ~(subnet[x]);
    broadcast[x] |= localip[x] & subnet[x];
  }

  printf("UDP ready: netmask %s broadcast %s\n",
    subnet.toString().c_str(),
    broadcast.toString().c_str()
  );

  // Send "Ready:\n"
  udp.beginPacket(broadcast,9000);
  udp.write((uint8_t const*)"Ready:\n",7);
  udp.endPacket();

  for (;;) {
    xEventGroupWaitBits(
      hevt,           // handle
      LED_CHG,        // bits to wait for
      pdTRUE,         // clear bits
      pdFALSE,        // wait for all bits
      portMAX_DELAY); // timeout
    char temp[16];

    // Send UDP packet:
    udp.beginPacket(broadcast,9000);
    for ( short x=0; x<N_LED; ++x ) {
      bool state = !!digitalRead(leds[x]);
      snprintf(temp,sizeof temp,
        "LED%d=%d\n",
        x,state);
      udp.write((uint8_t const*)temp,strlen(temp));
    }
    udp.write((uint8_t const*)"--\n",3);
    udp.endPacket();
  }
}

static void init_http() {
  unsigned count = 0;

  printf("WiFi connecting to SSID: %s\n",ssid);
  WiFi.begin(ssid,passwd);

  while ( WiFi.status() != WL_CONNECTED ) {
    delay(250);
    if ( ++count < 80 )
      printf(".");
    else {
      printf("\n");
      count = 0;
    }
  }

  printf("\nWiFi connected as %s\n",
    WiFi.localIP().toString().c_str());
  http.begin();

  xEventGroupSetBits(hevt,WIFI_RDY);
}

void setup() {
  int app_cpu = xPortGetCoreID();
  BaseType_t rc;

  // Configure LED GPIOs
  for ( int x=0; x<N_LED; ++x ) {
    pinMode(leds[x],OUTPUT);
    digitalWrite(leds[x],LOW);
  }

  // Create Event Group
  hevt = xEventGroupCreate();
  assert(hevt);

  // Allow USB to connect
  delay(2000);
  printf("\neventgr.ino:\n");

  // HTTP Server Task
  rc = xTaskCreatePinnedToCore(
    http_server, // function
    "http",   // Name
    2100,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    nullptr,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);

  // UDP Broadcast Task
  rc = xTaskCreatePinnedToCore(
    udp_broadcast, // function
    "udp",    // Name
    2100,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    nullptr,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);

  // Start WiFi
  init_http();
}

void loop() {
  // Not used:
  vTaskDelete(nullptr);
}
```

Listing 12-1. Event Groups demonstration program eventgr.ino.

Synchronization

Coordinating several tasks to operate in unison is difficult. The Event group offers a solution using the xEventGroupSync() function. Let's examine the function call with a special focus on the arguments and how they are used in the call:

```
EventBits_t xEventGroupSync(
  EventGroupHandle_t xEventGroup,     // Handle
  const EventBits_t uxBitsToSet,      // Bits to set
  const EventBits_t uxBitsToWaitFor,  // Bits that must be set
  TickType_t xTicksToWait             // Timeout or portMAX_DELAY
);
```

The argument uxBitsToSet and uxBitsToWaitFor operate together as an atomic operation. For example, if uxBitsToSet is the value 0b0010 and the value of uxBitsToWaitFor is 0b0011, then both bit-0 and bit-1 must be true before the call will return. But as part of this call, we are setting bit-1, so we only need bit-0 in addition to succeed. When the call succeeds all bits waited for will be cleared upon return. The returned value will be the value *before* the bits were cleared (0b0011 in this example).

When using a timeout, you must check the returned value to see that it matches the value expected:

```
EventBits_t ev = xEventGroupSync(h,0b0010,0b0011,50);

if ( (ev & 0b0011) != 0b0011 )
  // timed out!
```

Demonstration 2

A simple demonstration synchronizing three tasks with the loopTask() is provided in listing 12-2 (program evsync.ino). Use the same breadboard setup as Figure 12-1 and no Serial Monitor is used this time. This program will blink the three LEDs from three separate tasks, in a tightly synchronized fashion. The fourth task, the loopTask, will signal when the LEDs should blink, in function loop():

```
#define EV_RDY        0b1000
#define EV_ALL        (EV_RDY|0b0111)
...
void loop() {
  delay(1000);
  xEventGroupSetBits(hevt,EV_RDY);
}
```

From the above, you can see that the loop() task will set event bit 3 (EV_RDY value 0b1000). Bits 0 through 2 will be set by the separate tasks. These LED tasks all share the same task function:

```
0014: static int leds[N_LED] =
0015:   { GPIO_LED1, GPIO_LED2, GPIO_LED3 };
0016:
0017: static void led_task(void *arg) {
0018:   unsigned ledx = (unsigned)arg;  // LED Index
0019:   EventBits_t our_ev = 1 << ledx; // Our event
0020:   EventBits_t rev;
0021:   TickType_t timeout;
0022:   unsigned seed = ledx;
0023:
0024:   assert(ledx < N_LED);
0025:
0026:   for (;;) {
0027:     timeout = rand_r(&seed) % 100 + 10;
0028:     rev = xEventGroupSync(
0029:       hevt,       // Group event
0030:       our_ev,     // Our bit to set
0031:       EV_ALL,     // All bits required
0032:       timeout);   // Timeout
0033:
0034:     if ( (rev & EV_ALL) == EV_ALL ) {
0035:       // Not timed out: blink LED
0036:       digitalWrite(leds[ledx],
0037:         !digitalRead(leds[ledx]));
0038:     }
0039:   }
0040: }
```

The LED index is passed into the led_task() function by abusing the void pointer in line 18. Once the LED index is known, the required bit pattern is computed in line 19 (value 0b0001 for ledx==0, 0b0010 for ledx==1, etc.)

Line 27 computes a randomized timeout value. This was done as extra proof that we can synchronize the tasks because of the xEventGroupSync() function. Otherwise, synchronization might accidentally work at first, until enough time has passed, perhaps taking hours to get out of synch.

The focus of this demonstration is in lines 28 to 32. The value our_ev is the bit pattern calculated for our respective LED task (line 19). This is the bit that is set (line 30), before waiting for all bits (line 31). The value of EV_ALL is the value 0b1111. In effect, this call to xEventGroupSync() will not successfully return unless all tasks have applied their respective bit *and* the loopTask() has set the EV_RDY (0b1000) bit. Any other returned value represents a timed out call (tested in line 34).

When the demonstration runs, the three LEDs should blink in unison, alternating between on and off at one-second intervals. A question that you might have, is just how synchronized are they? To the human eye, they look synchronized but in the signal realm, there is some time slop. If you have a scope, measure and compare the three LED outputs. Figure 12-3 is a sample capture.

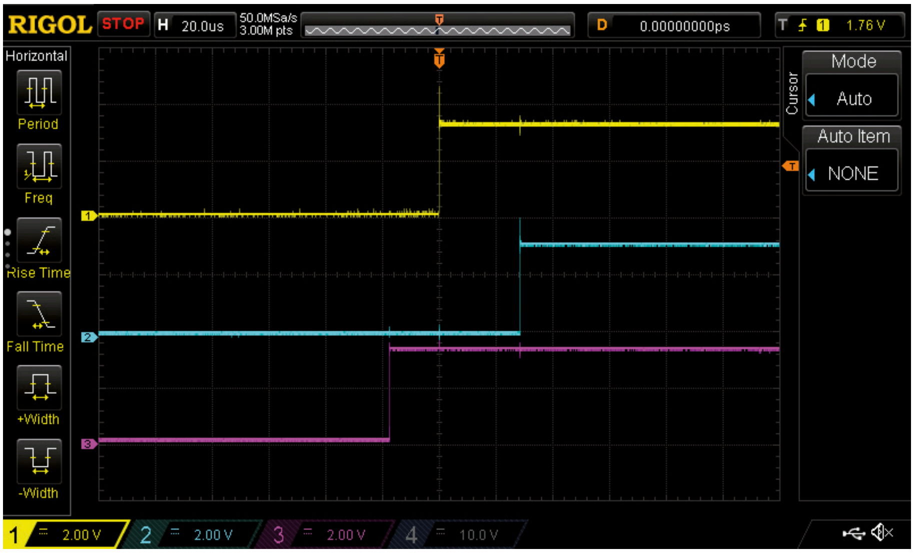
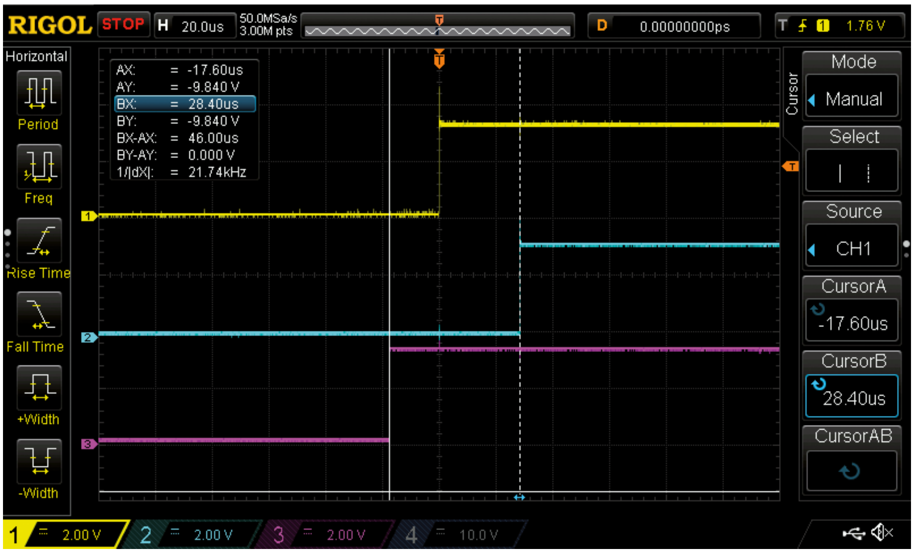


Figure 12-3. A sample capture of the synchronized LED outputs, with 20 µsec horizontal resolution.

At a 20 µsec horizontal resolution, it can be seen that the LED outputs are not perfectly synchronized. After all, each of our tasks are running from the same CPU, and only one task runs at a given instant. Based on the graticules, you can estimate the span, but Figure 12-4 provides another figure with the cursors enabled.

In the top left of the figure, the time reported for BX-AX (horizontal) is reported to be 46 µsec. Is this good enough? For the human eye, it is, but some applications may have more stringent requirements.



```
AX:    = -17.60us
AY:    = -9.840 V
BX:    = 28.40us
BY:    = -9.840 V
BX-AX: = 46.00us
BY-AY: = 0.000 V
1/|dX|: = 21.74kHz
```

STOP H 20.0us 50.0MSa/s 3.00M pts D 0.00000000ps T 1.76 V

Mode: Manual
Source: CH1
CursorA: -17.60us
CursorB: 28.40us

1 2.00 V 2 2.00 V 3 2.00 V 4 10.0 V

Figure 12-4. A sample capture of the synchronized LED outputs, with horizontal cursors on, showing 46 μsec of slop.

```
// evsync.ino

// LED GPIOs:
#define GPIO_LED1     18
#define GPIO_LED2     19
#define GPIO_LED3     21

#define N_LED         3

#define EV_RDY        0b1000
#define EV_ALL        (EV_RDY|0b0111)

static EventGroupHandle_t hevt;
static int leds[N_LED] =
  { GPIO_LED1, GPIO_LED2, GPIO_LED3 };

static void led_task(void *arg) {
  unsigned ledx = (unsigned)arg;  // LED Index
  EventBits_t our_ev = 1 << ledx; // Our event
  EventBits_t rev;
  TickType_t timeout;
  unsigned seed = ledx;

  assert(ledx < N_LED);

  for (;;) {
    timeout = rand_r(&seed) % 100 + 10;
    rev = xEventGroupSync(
      hevt,       // Group event
      our_ev,     // Our bit to set
      EV_ALL,     // All bits required
      timeout);   // Timeout

    if ( (rev & EV_ALL) == EV_ALL ) {
      // Not timed out: blink LED
      digitalWrite(leds[ledx],
        !digitalRead(leds[ledx]));
    }
  }
}

void setup() {
  int app_cpu = xPortGetCoreID();
  BaseType_t rc;

  // Create Event Group
  hevt = xEventGroupCreate();
  assert(hevt);

  // Configure LED GPIOs
  for ( int x=0; x<N_LED; ++x ) {
    pinMode(leds[x],OUTPUT);
    digitalWrite(leds[x],LOW);

    rc = xTaskCreatePinnedToCore(
      led_task, // function
      "ledtsk", // Name
      2100,     // Stack size
      (void*)x, // Parameters
      1,        // Priority
      nullptr,  // handle
      app_cpu   // CPU
    );
    assert(rc == pdPASS);
  }
}

void loop() {
  delay(1000);
  xEventGroupSetBits(hevt,EV_RDY);
}
```

Listing 12-2.

Auxiliary Functions

There are other event group functions that may be used in special situations. The first worthy mention is the vEventGroupDelete() function.

vEventGroupDelete()

Unlike some other FreeRTOS resources, this function can safely be called, when tasks are still blocked waiting for the event group being deleted.

```
void vEventGroupDelete(EventGroupHandle_t xEventGroup);
```

Tasks that are still blocked on the event group being deleted, will return the value zero when the event group is deleted.

Note: Code should not reuse the handle of the deleted event group. One solution would be to set the referencing handle to nullptr (NULL) after the delete. Then test the handle before attempting to use it.

xEventGroupClearBits()

The xEventGroupClearBits and xEventGroupClearBitsFromISR() functions can be used to clear event group bits. These two functions operate by sending a message to the RTOS daemon task, which executes at the priority of configTIMER_TASK_PRIORITY. For the ESP32 Arduino, this is defined as priority 1.

```
EventBits_t xEventGroupClearBits(
  EventGroupHandle_t xEventGroup,
  const EventBits_t uxBitsToClear
);

BaseType_t xEventGroupClearBitsFromISR(
  EventGroupHandle_t xEventGroup,
  const EventBits_t uxBitsToClear
);
```

Because these functions occur at the priority of the RTOS daemon task, you must ensure that your use of tasks priorities allow for it to execute from time to time. It must also be understood that these actions may not occur immediately due to being queued and executed at low priority.

xEventGroupGetBits()

The xEventGroupGetBits() and xEventGroupGetBitsFromISR() functions permit the task and ISR respectively to query the current event group bit values.

```
EventBits_t xEventGroupGetBits(EventGroupHandle_t xEventGroup);

EventBits_t xEventGroupGetBitsFromISR(EventGroupHandle_t xEventGroup);
```

xEventGroupSetBitsFromISR()

Interrupt service routines can also set event group bits using the xEventGroupSetBits-FromISR() function:

```
BaseType_t xEventGroupSetBitsFromISR(
  EventGroupHandle_t xEventGroup,
  const EventBits_t uxBitsToSet,
  BaseType_t *pxHigherPriorityTaskWoken
);
```

This permits ISRs to communicate events to multiple tasks in the same manner that the function xEventGroupSetBits() does.

Summary

With the use of xEventGroupWaitBits(), it was demonstrated that multiple tasks could block their execution until a barrier was released (WiFi ready). The same event group was able to notify another task when LED states changed. The final demonstration proved how an event group could be used to synchronize tasks. All of this proves the flexibility and utility of the FreeRTOS event group.

Exercises

1. How many event bits are available in the data type EventBits_t?
2. Can you use xEventGroupSetBits() and xEventGroupWaitBits() to synchronize instead of using the single function xEventGroupSync()?
3. How does an event group bit get used as a barrier?
4. How does a non-barrier event bit differ from the barrier type?
5. How many tasks can block on an event group?
6. What happens to the event group handle after the event group has been deleted by vEventGroupDelete()?
7. How do you distinguish the difference between a timed out xEventGroupWaitBits() call and a successful one?
8. What best describes the logical function of xWaitForAllBits when set to pdFALSE: OR or AND of the required uxBitsToWaitFor?

Web Resources

[1] This project was inspired by
https://randomnerdtutorials.com/esp32-web-server-arduino-ide/

Chapter 13 • Advanced Topics

Sometimes you need more than basic.

This chapter includes some miscellaneous content related to FreeRTOS that didn't fit into a chapter on its own. Three important topics including watchdog timers, critical sections, and task local storage are covered in this chapter.

Watchdog Timers

Some applications need to be constantly available. Sometimes the application might hang or fail due to components or libraries outside of your control. If the device runs in a remote place, then visits to the site to reset it are highly undesirable. Or perhaps the device is a flight control system for a drone and seizure in the air would have disastrous consequences. A watchdog timer can force a reset when configured events don't occur as expected.

The watchdog timer works on the following basic principles:

1. A timer is started.
2. As part of the operation of the application, the timer is reset periodically.
3. If the timer reaches a preconfigured value, the alarm is raised and the system panics.

When everything is working correctly, the timer is reset before it reaches the panic point. The process is like a water bucket for a leaky roof. When the application is working correctly, the bucket gets emptied periodically before it can get full. If however, the bucket overflows, then a panic procedure begins.

A common and effective panic procedure is to reset the device. This handles situations like a hung peripheral device where a reset and restart is likely to clear the problem. The alternative to this is to raise an alarm and take some other corrective action.

Watchdog Timer for the loopTask

The ESP32 Arduino environment provides some of the framework for the loopTask watchdog timer. Listing 13-1 illustrates the complete main.cpp that all ESP32 Arduino programs link with (at least when the macro CONFIG_AUTOSTART_ARDUINO is configured).

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_task_wdt.h"
```

```
0004: #include "Arduino.h"
0005:
0006: TaskHandle_t loopTaskHandle = NULL;
0007:
0008: #if CONFIG_AUTOSTART_ARDUINO
0009:
0010: bool loopTaskWDTEnabled;
0011:
0012: void loopTask(void *pvParameters)
0013: {
0014:     setup();
0015:     for(;;) {
0016:         if(loopTaskWDTEnabled){
0017:             esp_task_wdt_reset();
0018:         }
0019:         loop();
0020:     }
0021: }
0022:
0023: extern "C" void app_main()
0024: {
0025:     loopTaskWDTEnabled = false;
0026:     initArduino();
0027:     xTaskCreateUniversal(loopTask, "loopTask",
           8192, NULL, 1, &loopTaskHandle,
           CONFIG_ARDUINO_RUNNING_CORE);
0028: }
0029:
0030: #endif
```

Listing 13-1. The Arduino startup module main.cpp.

From the listing, it can be seen that the watchdog timer reset call (lines 16 and 17) is disabled by default (line 25) because the global loopTaskWDTEnabled is initialized to false (line 25). The loop() function is called repeatedly from the forever loop in lines 15 to 20. When the watchdog *is enabled*, a call is made to esp_task_wdt_reset() prior to each call to the function loop().

From the listing it is tempting to think that all you need to do to enable the watchdog is to set the global to enable it:

```
loopTaskWDTEnabled = true;
```

There is, however, more to enabling the task watchdog. What this does accomplish however, is to have the loopTask() invoke esp_task_wdt_reset() for each loop iteration.

Enabling Task Watchdog

To make use of the task watchdog, a few ESP32 specific calls need to be made in addition to enabling the global boolean loopTaskWDTEnabled. The demonstration program watchdog1.ino is provided in Listing 13-2.[1] Note that an include file for "esp_task_wdt.h" must also be provided in the program (line 3).

```
0001: // watchdog1.ino
0002:
0003: #include <esp_task_wdt.h>
0004:
0005: extern bool loopTaskWDTEnabled;
0006: static TaskHandle_t htask;
0007:
0008: void setup() {
0009:   esp_err_t er;
0010:
0011:   htask = xTaskGetCurrentTaskHandle();
0012:   loopTaskWDTEnabled = true;
0013:   delay(2000);
0014:
0015:   er = esp_task_wdt_status(htask);
0016:   assert(er == ESP_ERR_NOT_FOUND);
0017:
0018:   if ( er == ESP_ERR_NOT_FOUND ) {
0019:     er = esp_task_wdt_init(5,true);
0020:     assert(er == ESP_OK);
0021:     er = esp_task_wdt_add(htask);
0022:     assert(er == ESP_OK);
0023:     printf("Task is subscribed to TWDT.\n");
0024:   }
0025: }
0026:
0027: static int dly = 1000;
0028:
0029: void loop() {
0030:   esp_err_t er;
0031:
0032:   printf("loop(dly=%d)..\n",dly);
0033:   er = esp_task_wdt_status(htask);
0034:   assert(er == ESP_OK);
0035:   delay(dly);
0036:   dly += 1000;
0037: }
```

Listing 13-2. The demonstration program watchdog1.ino.

Lines 15 and 16 of the setup() routine just prove to us that the task watchdog timer has not been established for the loopTask (this can be left out of a finished program). Line 18 tests for this, and invokes lines 19 to 23 if the task is not registered for watchdog events. The call to esp_task_wdt_init() in line 19 sets the timeout (5 seconds) and requests that the watchdog perform a panic reboot if the watchdog is triggered (argument two is true). The call to esp_task_wdt_add() adds the task handle to the list of handles that the watchdog will listen for. In this example, only one handle is registered. When multiple handles are registered, they must *all* make a call to reset the watchdog. From this point on, the task watchdog will monitor for resets of the form:

```
esp_task_wdt_reset();  // Assumes the calling task
```

Compile and flash the program watchdog1.ino, with the Serial Monitor ready:

```
Task is subscribed to TWDT.
loop(dly=1000)..
loop(dly=2000)..
loop(dly=3000)..
loop(dly=4000)..
loop(dly=5000)..
E (34223) task_wdt: Task watchdog got triggered. The following tasks did not
reset the watchdog in time:
E (34223) task_wdt:  - loopTask (CPU 1)
E (34223) task_wdt: Tasks currently running:
E (34223) task_wdt: CPU 0: IDLE0
E (34223) task_wdt: CPU 1: IDLE1
E (34223) task_wdt: Aborting.
abort() was called at PC 0x400d136f on core 0

Backtrace: 0x4008b00c:0x3ffbe160 0x4008b239:0x3ffbe180 0x400d136f:0x3ffbe1a0
0x40084109:0x3ffbe1c0 0x400e80f7:0x3ffbbff0 0x400d1b4f:0x3ffbc010
0x400891b2:0x3ffbc030 0x40087cc1:0x3ffbc050

Rebooting...
ets Jun  8 2016 00:22:57

rst:0xc (SW_CPU_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
```

```
entry 0x400806b8
Task is subscribed to TWDT.
loop(dly=1000)..
loop(dly=2000)..
...
```

The demo program uses a delay of 1000 milliseconds initially but increases that by 1000 milliseconds with each call into loop(). Once the task watchdog timer senses a lack of response within 5 seconds (see line 19), it panics and reboots the ESP32. As part of the panic report, we see that "loopTask (CPU 1) " is failing to reset the watchdog timer in time.

Note: The first argument to esp_task_wdt_init() is in units of *seconds* – not milliseconds. It is easy to assume milliseconds when its use within FreeRTOS is so prevalent. If you fail to get a watchdog reset in your application (when expected), check that you've not made this type of mistake.

Watchdog For Multiple Tasks

The ESP32 can monitor multiple tasks with a watchdog. However, there can only be one master timeout value (in seconds) and panic status configured. When multiple tasks are registered, the watchdog timer is triggered if *any* of the registered tasks fails to reset on time. Listing 13-3 is a listing of watchdog2.ino to demonstrate this,[2] which uses one LED to give task2 some purpose. The schematic is shown in Figure 13-1.

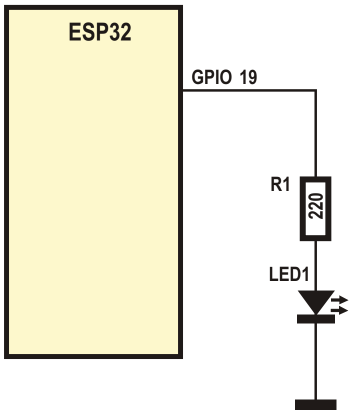


Figure 13-1. The schematic for the watchdog2.ino demonstration.

Unlike the prior example, task2() will have to make a call to esp_task_wdt_reset() on behalf of itself. The loopTask did this automatically for it (Listing 13-1 line 17), but other tasks must issue the call for itself. In Listing 13-3, this is done for task2() in line 20.

For this demo, the hall effect sensor is read in line 12 of task2(). This is done so that the random number seed (line 12) can be initialized with a different value for each time that the ESP32 is started. Otherwise, the demonstration would just repeat the events of the previous run.

```
// watchdog2.ino

#include <esp_task_wdt.h>

#define GPIO_LED    19

extern bool loopTaskWDTEnabled;
static TaskHandle_t htask;

static void task2(void *arg) {
  esp_err_t er;
  unsigned seed = hallRead();

  er = esp_task_wdt_add(nullptr);
  assert(er == ESP_OK);

  for (;;) {
    digitalWrite(GPIO_LED,
      1 ^ !!digitalRead(GPIO_LED));
    esp_task_wdt_reset();
    delay(rand_r(&seed)%7*1000);
  }
}

void setup() {
  int app_cpu = xPortGetCoreID();
  BaseType_t rc;
  esp_err_t er;

  pinMode(GPIO_LED,OUTPUT);
  digitalWrite(GPIO_LED,LOW);

  htask = xTaskGetCurrentTaskHandle();
  loopTaskWDTEnabled = true;
  delay(2000);

  er = esp_task_wdt_status(htask);
  assert(er == ESP_ERR_NOT_FOUND);

  if ( er == ESP_ERR_NOT_FOUND ) {
    er = esp_task_wdt_init(5,true);
    assert(er == ESP_OK);
    er = esp_task_wdt_add(htask);
    assert(er == ESP_OK);
  }

  rc = xTaskCreatePinnedToCore(
    task2,    // function
    "task2",  // Name
    2000,     // Stack size
    nullptr,  // Parameters
    1,        // Priority
    nullptr,  // handle
    app_cpu   // CPU
  );
  assert(rc == pdPASS);
}

static int dly = 1000;

void loop() {
  esp_err_t er;

  printf("loop(dly=%d)..\n",dly);
  er = esp_task_wdt_status(htask);
  assert(er == ESP_OK);
  delay(dly);
  dly += 1000;
}
```

Listing 13-3. The watchdog2.ino demonstration program using two registered watchdog tasks.

In the following example session, the loopTask prints to the Serial Monitor until after it reports dly=5000. At that point, the watchdog triggers because task2() didn't meet the timing requirements. After the reboot, the loopTask eventually triggers the watchdog in the second run. Rebooting still again, it is shown that both the loopTask and task2 are blamed by the watchdog. Because the actions in this demonstration depend upon random delays, you are likely to experience a different session output when you run it.

```
loop(dly=1000)..
loop(dly=2000)..
loop(dly=3000)..
loop(dly=4000)..
loop(dly=5000)..
E (32226) task_wdt: Task watchdog got triggered. The following tasks did not
reset the watchdog in time:
E (32226) task_wdt:  - task2 (CPU 1)
E (32226) task_wdt: Tasks currently running:
E (32226) task_wdt: CPU 0: IDLE0
E (32226) task_wdt: CPU 1: IDLE1
E (32226) task_wdt: Aborting.
```

```
abort() was called at PC 0x400d2bef on core 0

Backtrace: 0x4008b680:0x3ffbe170 0x4008b8ad:0x3ffbe190 0x400d2bef:0x3ffbe1b0
0x4008477d:0x3ffbe1d0 0x400e844b:0x3ffbbff0 0x400d33cf:0x3ffbc010
0x40089826:0x3ffbc030 0x40088335:0x3ffbc050

Rebooting...
ets Jun  8 2016 00:22:57

rst:0xc (SW_CPU_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
loop(dly=1000)..
loop(dly=2000)..
loop(dly=3000)..
loop(dly=4000)..
loop(dly=5000)..
E (34226) task_wdt: Task watchdog got triggered. The following tasks did not
reset the watchdog in time:
E (34226) task_wdt:  - loopTask (CPU 1)
E (34226) task_wdt: Tasks currently running:
E (34226) task_wdt: CPU 0: IDLE0
E (34226) task_wdt: CPU 1: IDLE1
E (34226) task_wdt: Aborting.
abort() was called at PC 0x400d2bef on core 0

Backtrace: 0x4008b680:0x3ffbe170 0x4008b8ad:0x3ffbe190 0x400d2bef:0x3ffbe1b0
0x4008477d:0x3ffbe1d0 0x400e844b:0x3ffbbff0 0x400d33cf:0x3ffbc010
0x40089826:0x3ffbc030 0x40088335:0x3ffbc050

Rebooting...
ets Jun  8 2016 00:22:57

rst:0xc (SW_CPU_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
```

```
load:0x3fff001c,len:1216
ho 0 tail 12 room 4
load:0x40078000,len:9720
ho 0 tail 12 room 4
load:0x40080400,len:6352
entry 0x400806b8
loop(dly=1000)..
loop(dly=2000)..
loop(dly=3000)..
loop(dly=4000)..
loop(dly=5000)..
E (34226) task_wdt: Task watchdog got triggered. The following tasks did not
reset the watchdog in time:
E (34226) task_wdt:  - loopTask (CPU 1)
E (34226) task_wdt:  - task2 (CPU 1)
E (34226) task_wdt: Tasks currently running:
E (34226) task_wdt: CPU 0: IDLE0
E (34226) task_wdt: CPU 1: IDLE1
E (34226) task_wdt: Aborting.
abort() was called at PC 0x400d2bef on core 0

Backtrace: 0x4008b680:0x3ffbe170 0x4008b8ad:0x3ffbe190 0x400d2bef:0x3ffbe1b0
0x4008477d:0x3ffbe1d0 0x400e844b:0x3ffbbff0 0x400d33cf:0x3ffbc010
0x40089826:0x3ffbc030 0x40088335:0x3ffbc050

Rebooting...
```

Non-Arduino Watchdog Use

If you only use the task watchdog timer from other tasks rather than the loopTask, there is no need to set the Arduino global:

```
loopTaskWDTEnabled = true;
```

This global variable only affects the loopTask if statement in line 16 of the loopTask:

```
0012: void loopTask(void *pvParameters)
0013: {
0014:     setup();
0015:     for(;;) {
0016:         if(loopTaskWDTEnabled){
0017:             esp_task_wdt_reset();
0018:         }
0019:         loop();
0020:     }
0021: }
```

In other words, this does not affect other tasks that may be registered for the watchdog facility.

The Idle Task

In the ESP-IDF environment, the Idle task uses the watchdog timer by default, unless configured otherwise. In the Arduino environment, this is *not* the case. This can be checked in code, assuming the code is running in the application CPU (1 for dual-core):

```
TaskHandle_t h = xTaskGetIdleTaskHandle();
esp_err_t e = esp_task_wdt_status(h);

if ( e == ESP_OK )
  // watchdog timer is in effect for Idle task
else
  // watchdog timer is not in effect for Idle task
```

Monitoring the Idle task is a good idea because it will catch task scheduling and priority problems. Some functions including FreeRTOS timers run from the RTOS daemon (idle) task, which run at priority zero. If other tasks at higher priorities are consuming all of the CPU time, then actions performed by the Idle task cannot occur. Enabling the watchdog can make this problem obvious by performing a watchdog reset when it is detected. To enable the watchdog for the Idle task in Arduino, perform the following. This assumes the code is running from the application CPU, which it always will be in function setup():

```
TaskHandle_t h = xTaskGetIdleTaskHandle();
esp_err_t e = esp_task_wdt_status(h);

if ( e != ESP_OK ) {
  e = esp_task_wdt_add(h); // Add Idle task
  assert(e == ESP_OK);
}
```

If the returned status is, in fact, ESP_OK, then it may be because your Arduino is running on a single-core instance for the ESP32, which already has a watchdog running to support WiFi, etc.

Critical Sections

There are sometimes points in the application program where you cannot have the execution of the code interrupted by the scheduler or processing in an ISR. For example, if you were to use the ESP32 peripheral registers to drive three GPIO outputs simultaneously, you might have to perform a load, OR bit-wise operation, and then store to a register. If your code were to be interrupted before the operation was completed, then the input values may change, invalidating the result. The critical section allows for a small amount of processing to occur uninterrupted.

In *generic* FreeRTOS, the critical sections are managed by the following pair of FreeRTOS macros:

```
taskENTER_CRITICAL();
  ...
taskEXIT_CRITICAL();
```

On many FreeRTOS platforms, this is implemented by disabling interrupts. But the dual-core ESP32 complicates this simple approach.

ESP32 Critical Sections
Because the ESP32 has two CPU cores – disabling interrupts on one CPU does not affect the other. The Espressif solution is to provide a modified pair of macros using an ESP32 system mutex:

```
portMUX_TYPE mutex = portMUX_INITIALIZER_UNLOCKED;

portENTER_CRITICAL(&mutex);
  ...
portEXIT_CRITICAL(&mutex);
```

The design of this approach is such that when entering a critical section, interrupts are disabled like the generic FreeRTOS implementation but will also have the calling core take the spin lock internal to the mutex. In this manner, the other core is left unaffected by the critical section. If however, the other core attempts to enter the same critical section and take the same mutex, it will spin until the mutex (and its spin lock) is released by the first core. Note that the FreeRTOS scheduler for the current core is disabled by the side-effect of disabled interrupts (the system tick interrupt does not occur).

These two macros must be paired but can be nested. Nesting allows safety to be achieved in called functions. Like ISR code, the code in the critical section should be kept as short as possible to prevent interference with important interrupts. Listing 13-4 provides a listing of program critical.ino.[3]

```
// critical.ino

// LED GPIOs:
#define GPIO_LED1     18
#define GPIO_LED2     19
#define GPIO_LED3     21

void setup() {
  // Configure LED GPIOs
  pinMode(GPIO_LED1,OUTPUT);
  pinMode(GPIO_LED2,OUTPUT);
  pinMode(GPIO_LED3,OUTPUT);
  digitalWrite(GPIO_LED1,LOW);
  digitalWrite(GPIO_LED2,LOW);
  digitalWrite(GPIO_LED3,LOW);
}

void loop() {
  static portMUX_TYPE mutex =
    portMUX_INITIALIZER_UNLOCKED;
  bool state = !!digitalRead(GPIO_LED1) ^ 1;

  // Change 3 LEDs at once
  portENTER_CRITICAL(&mutex);
  digitalWrite(GPIO_LED1,state);
  digitalWrite(GPIO_LED2,state);
  digitalWrite(GPIO_LED3,state);
  portEXIT_CRITICAL(&mutex);

  delay(500);
}
```

Listing 13-4. The critical section demonstration program critical.ino.

When the demonstration runs, the three LEDs should blink in unison. Figure 13-2 provides a scope trace of the GPIO outputs, to measure the accuracy of synchronization. The trace uses a horizontal resolution of 100 ns. From this you can see that there is a total slop of about 200 ns between the first and the last LED being set high. The critical section ensures that this will always be the case, without any interruption due to interrupts or preemptive task execution.

Note: Improved synchronization can be achieved with the GPIO outputs if the program were to take advantage of direct control of the ESP32 peripheral registers. Using the peripheral registers, all three LEDs can be changed at once.

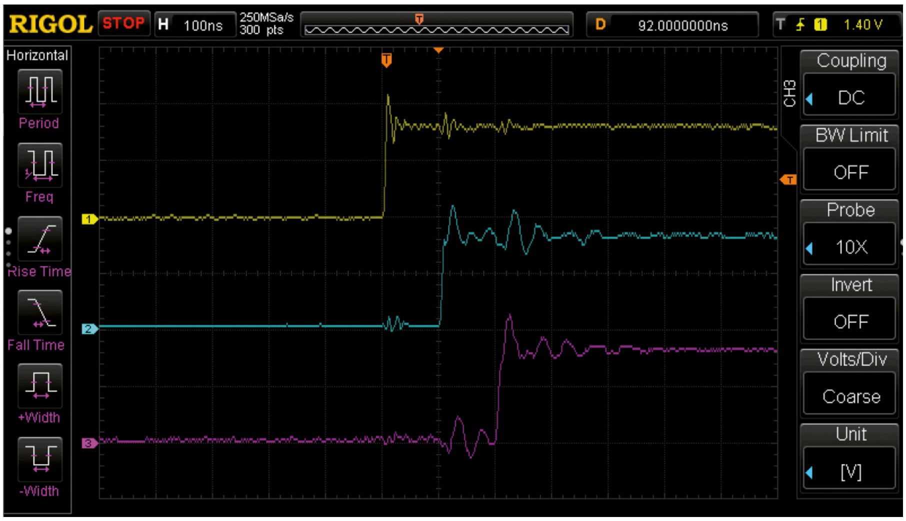


Figure 13-2. Scope trace of the three output LED GPIO pins, horizontal resolution is 100 ns.

Critical Sections for ISRs

Even though the ESP32 implements these ISR specific macros in terms of the same non-ISR macros, you should always use the ISR specific function names. This helps the maintainer of your code and makes it more portable to future platforms that your code might be ported to. Notice the _ISR suffixes added to the macro names:

```
portMUX_TYPE mutex = portMUX_INITIALIZER_UNLOCKED;

portENTER_CRITICAL_ISR(&mutex);
  ...
portEXIT_CRITICAL_ISR(&mutex);
```

Interrupts

If you need to disable interrupts (for the current ESP32 core), the following macros can be used. However, you should likely be using a critical section instead:

```
void portDISABLE_INTERRUPTS();
void portENABLE_INTERRUPTS();
```

Task Local Storage

One of the scourges of COBOL before the object-oriented version of the language was developed was that every data item was visible to the entire source module. This was like having all of your variables declared globally in a C/C++ program. There are also reasons why FreeRTOS tasks need task-specific storage when multiple instances of a task must run with different data. Rather than require the application programmer to create a table or use a C++ std::map, FreeRTOS provides facilities to store and retrieve pointers associated with the task itself. FreeRTOS provides the following API functions, which are available to the ESP32 Arduino programmer:

```
        void vTaskSetThreadLocalStoragePointer(
          TaskHandle_t xTaskToSet,
          BaseType_t xIndex,
          void *pvValue
        );

        void *pvTaskGetThreadLocalStoragePointer(
          TaskHandle_t xTaskToQuery,
          BaseType_t xIndex
        );
```

The task handle can be nullptr (NULL) when the task is referencing its local storage pointers. Otherwise, the referenced task's handle is supplied in the first argument.

The second argument (xIndex) *must be zero* for the ESP32 Arduino because the Arduino environment is built configured for only one pointer. Other FreeRTOS environments, may have more.

When setting the pointer, the third argument (pvValue) points to the task-specific data. When getting the pointer, the pointer is the return value.

Listing 13-5 provides a listing of the tasklocal.ino demonstration program,[4] with its schematic in Figure 13-3. The program uses three tasks to blink the three LEDs at *different independent rates* but uses task local storage to manage its configuration and state.

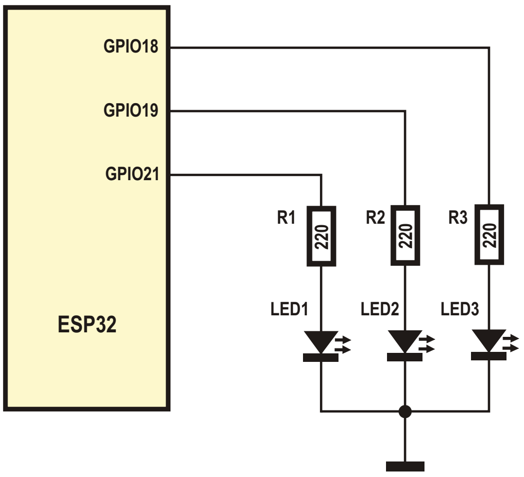


Figure 13-3. Schematic for tasklocal.ino demonstration program.

```
// tasklocal.ino

// LED GPIOs:
#define GPIO_LED1   18
#define GPIO_LED2   19
#define GPIO_LED3   21

#define N_LED       3

static int leds[N_LED] =
  { GPIO_LED1, GPIO_LED2, GPIO_LED3 };

struct s_task_local {
  int     index;
  int     led_gpio;
  bool    state;
};

static void blink_led() {
  s_task_local *plocal = (s_task_local*)
    pvTaskGetThreadLocalStoragePointer(nullptr,0);

  delay(plocal->index*250+250);
  plocal->state ^= true;
  digitalWrite(plocal->led_gpio,plocal->state);
}

static void led_task(void *arg) {
  int x = (int)arg;
  s_task_local *plocal = new s_task_local;

  plocal->index = x;
  plocal->led_gpio = leds[x];
  plocal->state = false;

  pinMode(plocal->led_gpio,OUTPUT);
  digitalWrite(plocal->led_gpio,LOW);

  vTaskSetThreadLocalStoragePointer(
    nullptr,
    0,
    plocal);

  for (;;) {
    blink_led();
  }
0047: }
0048:
0049: void setup() {
0050:   int app_cpu = xPortGetCoreID();
0051:   BaseType_t rc;
0052:
0053:   for ( int x=0; x<N_LED; ++x ) {
0054:     rc = xTaskCreatePinnedToCore(
0055:       led_task, // function
0056:       "ledtsk", // Name
0057:       2100,     // Stack size
0058:       (void*)x, // Parameters
0059:       1,        // Priority
0060:       nullptr,  // handle
0061:       app_cpu   // CPU
0062:     );
0063:     assert(rc == pdPASS);
0064:   }
0065: }
0066:
0067: void loop() {
0068:   vTaskDelete(nullptr);
0069: }
```

Listing 13-5. The tasklocal.ino demonstration of task local storage.

The task sets up its own local storage in lines 30 to 42. Line 30 allocates the structure s_task_local, and initializes its members in lines 32 to 34. Once that is done, the forever loop in lines 44 to 46 are executed. Within that loop, function blink_led() can be called without parameters because it can obtain the task's local storage as needed (lines 20 and 21). This relies on the fact that nullptr for task handle causes the current task to be assumed.

```
0019: static void blink_led() {
0020:   s_task_local *plocal = (s_task_local*)
0021:     pvTaskGetThreadLocalStoragePointer(nullptr,0);
0022:
0023:   delay(plocal->index*250+250);
0024:   plocal->state ^= true;
0025:   digitalWrite(plocal->led_gpio,plocal->state);
0026: }
```

Once it has obtained the local task storage pointer, which is known to be a pointer to struct s_task_local in this program, it has everything it needs for lines 23 to 25.

uxTaskGetNumberOfTasks()
If your application needs to know how many tasks are currently executing, the FreeRTOS function uxTaskGetNumberOfTasks() will return the count:

```
UBaseType_t uxTaskGetNumberOfTasks(void);
```

xTaskGetSchedulerState()
For applications that may suspend the scheduler, the information function xTaskGetSchedulerState() may be useful:

```
BaseType_t xTaskGetSchedulerState(void);
```

Potential return values include:

- taskSCHEDULER_NOT_STARTED
- taskSCHEDULER_RUNNING
- taskSCHEDULER_SUSPENDED

eTaskGetState()
A task's state can be queried with the eTaskGetState() call:

```
eTaskState eTaskGetState(TaskHandle_t pxTask);
```

The possible values returned are:

- eRunning – task is querying its state (or running on a different core)
- eBlocked
- eSuspended
- eDeleted – waiting for the RTOS daemon task to clean up its memory resources.

xTaskGetTickCount()
If your timer needs are simple, you can make use of the system ticks counter. Note that this count value can wrap around after it overflows:

```
TickType_t xTaskGetTickCount(void);
TickType_t xTaskGetTickCountFromISR(void);
```

Calling these functions at different times it is possible to compute the elapsed time in ticks.

vTaskSuspendAll()
If you want to suspend and resume the scheduler (for the current core only), the following functions can be used:

```
void vTaskSuspendAll(void);

BaseType_t xTaskResumeAll(void);
```

The xTaskResumeAll() function returns the following values:

- pdTRUE – The scheduler was transitioned into the active state and caused a pending context switch to occur.
- pdFALSE – Either the scheduler transitioned into the active state without causing a context switch, or the scheduler was left in the suspended state due to nested calls.

ESP32 Arduino Limitations

Other FreeRTOS functions are not currently supported by the ESP32 Arduino environment. Some of them likely have been omitted to save space given their limited interest to the Arduino community. Functions like xTaskCallApplicationTaskHook() is one example of advanced functionality that has limited use. Those who want to more fully explore FreeRTOS may want to install and use the ESP-IDF development framework.[5]

Summary

This chapter has examined some of the more advanced concepts and API within the ESP32 Arduino FreeRTOS context including watchdog timers and critical sections. There are several other advanced FreeRTOS functions that apply to certain platforms. For a complete reference on the API, you are encouraged to download and peruse the PDF FreeRTOS Reference manual.[6]

Exercises

1. What is the purpose of a critical section?
2. When the software is correct, when is a watchdog timer useful?
3. Why enable the watchdog timer for the Idle task?
4. Does setting global boolean loopTaskWDTEnabled to true enable the watchdog timer for the loopTask?
5. Why use task local storage?
6. Can the current task continue to execute after the call to vTaskSuspendAll() is called?
7. What unit of time does xTaskGetTickCount() use?
8. When using xTaskGetTickCount() for computing elapsed time, what do you need to be careful for?

Web Resources

[1] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/watchdog1/watchdog1.ino
[2] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/watchdog2/watchdog2.ino
[3] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/critical/critical.ino
[4] https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/tasklocal/tasklocal.ino
[5] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html
[6] https://www.freertos.org/wp-content/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

Chapter 14 • Gatekeeper Tasks

Q: Why did the two gatekeepers start fighting?
A: Because they were fencing.

A large number of FreeRTOS API functions have been presented in this book as valid choices for managing the operational design of your application. But as the number of operating semaphores, mutexes, and queues, etc. increase, the design can reach a point where the approach seems difficult to manage. To make the design simpler to verify, it is sometimes better to divide and conquer at the system level. Moving functionality into a gatekeeper task allows the designer to perfect the gatekeeper and guarantee that the client tasks can't mess things up. This chapter provides a demonstrator gatekeeper task using two PCF8574P GPIO extender chips, similar to what was used in Chapter 8 Mutexes.

Gatekeepers

We often see examples of gatekeepers in daily life without giving it much thought. A common example is when you go to a movie or concert – you enter the venue through a gate controlled by the gatekeeper. Usually, there is a ticket required unless it is a free event. But even free events have gatekeepers to keep trouble-makers out and to check for factors affecting safety.

Software also requires certain levels of security and safety. On one hand, we might need to make sure that independent tasks don't clash in their use of the I2C bus, for example. There may also be a need to manage state for each of the I2C devices involved. Centralizing access to these bus devices through a gatekeeper task can make it much easier to provide proper order and access.

The Chapter 8 approach of using mutexes to protect resources is the non-gatekeeper approach. To verify that the non-gatekeeper program is safe and correct requires that you to locate all the access points and verify that the correct protocol has been observed. For large applications, this can become impractical. The gatekeeper approach, on the other hand, centralizes access. The gatekeeper API can serialize all interactions through a message queue to preserve order and safety. Because the code for the gatekeeper task is centralized, it is easier to test and verify with unit testing.

Demonstration
To demonstrate the utility of a gatekeeper task, the program gatekeeper.ino in Listing 14-1 is provided (almost any ESP32 can be used). The program will monitor three push buttons in a simple polling task (usr_task1) to activate the corresponding LEDs when pushed. A second task (usr_task2) will simply blink one LED on one of the PCF8574P devices. Thus we have an application with two independent tasks interacting with a common I2C bus and a pair of PCF8574P chips (Figure 14-1 for the pinout).

To make things more interesting and to prove the correctness of the gatekeeper code, one button is sensed on the second PCF8574P chip, while the other two button inputs come from the other. The three LEDs that are lit in response to the push buttons are driven by the first chip of the pair, while the blinking LED is driven by the second.

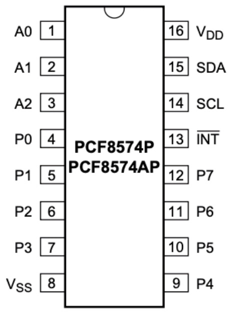


Figure 14-1. The 16-pin PCF8574P DIP pinout.

Extension GPIO Designations
One of the advantages of using a gatekeeper task is that we can define our interface to the two GPIO extender chips. Rather than require the caller to specify the I2C address and bit pattern for the device port data, we can map a virtual GPIO number to the actual I2C address and bit pattern internally to the gatekeeper code. To prevent confusion with ESP32 GPIO numbers, let’s use the name XGPIO in this chapter to refer to the PCF8574P GPIO pins.

Table 14-1 outlines the XGPIO to pin mappings used in this demonstration. For example, XGPIO 1 refers to the I/O port P1 on the *first* PCF8574P chip (address 0x20 or 0x38). XGPIO 15 refers to I/O port P7 on the *second* PCF8575P chip (address 0x21 or 0x39). Using this convention, it is a simple matter of specifying the XGPIO number to indicate the extender chip and pin.

XGPIO	PCF8574P I2C Address	PCF8574AP I2C Address	Pin Name	Pin Number
0	0x20	0x38	P0	4
1	0x20	0x38	P1	5
2	0x20	0x38	P2	6
...				
7	0x20	0x38	P7	12
8	0x21	0x39	P0	4
...				
15	0x21	0x39	P7	12

Table 14-1. XGPIO Mapping Convention used for gatekeeper.ino.

Gatekeeper API

The API that the demonstration gatekeeper task will support is simple. It consists of the following pair of function calls:

```
short pcf8574_get(uint8_t port);
short pcf8574_put(uint8_t port,bool value);
```

These functions fetch a bit value from the specified XGPIO (argument port) or write a bit to it respectively (argument value). The values returned are:

- 0 – get value read (or written)
- 1 – get value read (or written)
- -1 – I/O to PCF8574P device failed

Because the I2C transaction can fail, primarily due to a wiring error, the return function will return -1 when it fails.

Demonstration XGPIO

Now let's map out the resources that are used by the demonstration. Table 14-2 lists the C macro name used under the Macro column for each button and LED, referenced in the pair of PCF8574P chips. For example, BUTTON0 uses XGPIO 12, which is P4 on the second PCF8574P chip (device dev1), port P4, chip pin number 9. This push button, in turn, will drive LED0, on XGPIO 1, which drives P1 on the first chip (dev0), which is pin number 5. The LED3 is driven by a second task named usr_task2, which will blink at half-second intervals.

Macro	XGPIO	Port	Chip Pin	Macro	XGPIO	Port	Chip Pin
BUTTON0	12	P4 on dev1	9	LED0	1	P1 on dev0	5
BUTTON1	5	P5 on dev0	10	LED1	2	P2 on dev0	6
BUTTON2	4	P4 on dev0	9	LED2	3	P3 on dev0	7
usr_task2 blinks				LED3	8	P0 on dev1	4

Table 14-2. Table of extension GPIO inputs and Outputs

The schematic diagram is provided in Figure 14-2. Be sure to connect the address pins A0, A1, and A2 on each chip (these are easy to forget). The second PCF8574P chip must have A0 wired to the +3.3 volt line to register as a 1-bit. Notice also that the LEDs must have their cathodes connected to the GPIO pins so that the current is *sinked* instead of sourced (the PCF8574 only sources a maximum of 100 µA but can sink up to 25 mA). This means that the LEDs are active low in this configuration. Like Chapter 8, note that there are two varieties of the PCF8574 chip, with different I2C addresses. If you have the PCF8574A type, then change the following line in the program to a 1:

```
0007: // Set to 1 for PCF8574A
0008: #define PCF8574A      0
```

This demonstration doesn't make use of the Serial Monitor. However, if you have trouble getting it to work, be sure to start the Serial Monitor because an assertion may be triggered. The assertion fault will report the source line of the program that is failing. This will provide a clue as to what is failing (perhaps the I2C transaction fails because of its configured address or wiring problem).

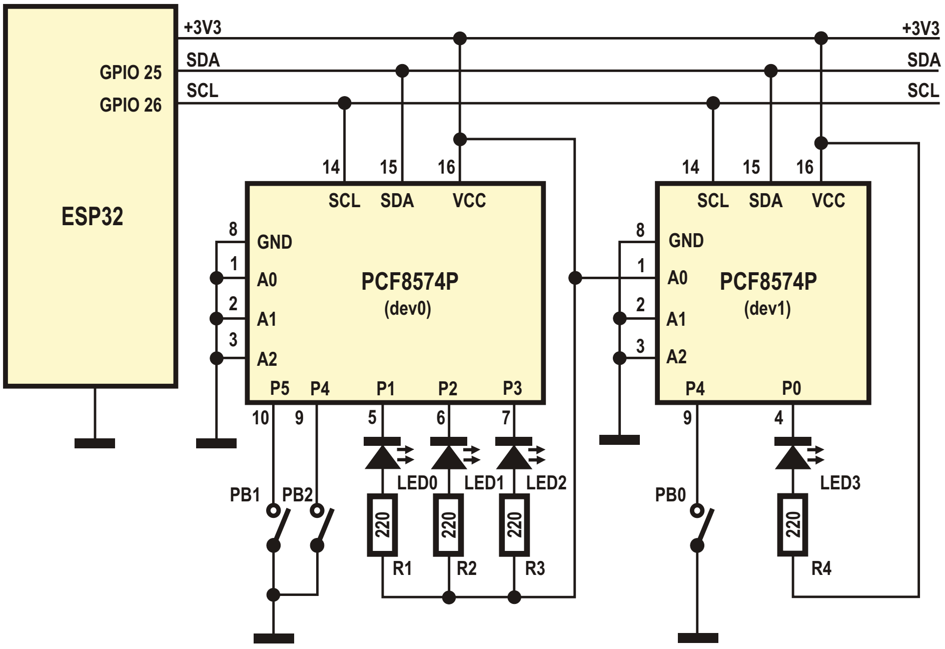


Figure 14-2. Schematic diagram for demonstration program gatekeeper.ino.

From a breadboard perspective, there is a fairly large number of Dupont/jumper wires needed. This can easily lead to wiring errors, especially if eyesight is a problem. Don't despair if it doesn't work at first – see the section Troubleshooting later on for some tips on things to check. It is most important that the chips are oriented the correct way round for the power connections. Applying reverse or incorrect power connections is often fatal for the chips.

Operation

Before we look at the code in detail, let's describe what the demonstration is intended to do. The push buttons (PB1, PB2, and PB3 in Figure 14-2) are the push buttons. Pushing any combination of those should result in LED0, LED1, and LED2 lighting respectively. The buttons can be held down in combination to light one or more LEDs. LED3 should immediately start blinking when the circuit is powered up and reset.

Figure 14-3 illustrates the author's breadboard layout. I used a convenient four-button PCB for the push buttons, shown at the bottom of the photo (only three of the buttons were used). The LEDs were all paired with a soldered on 220-ohm resistor. From the photo, you can see the number of jumper wires involved.

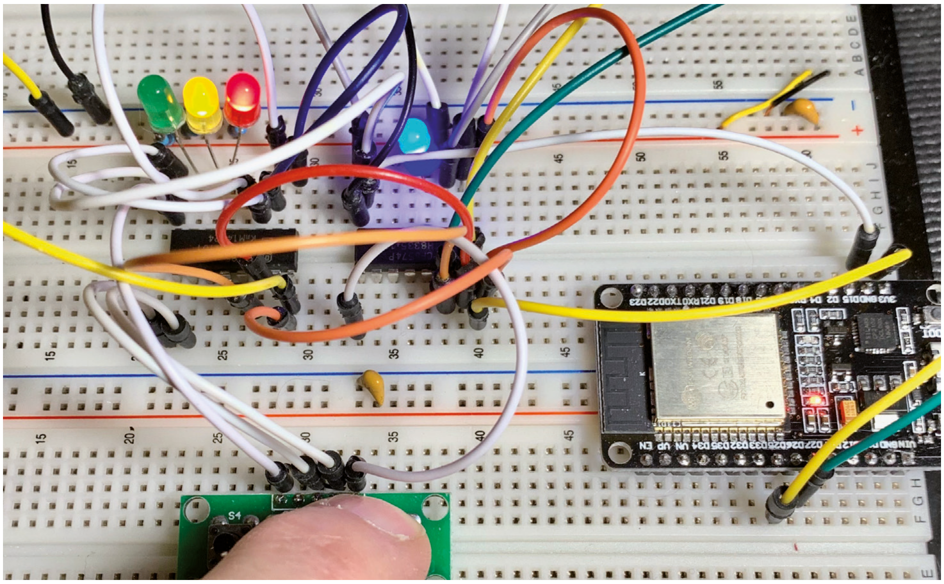


Figure 14-3. A breadboard layout of the gatekeeper.ino project and dev board ESP32.

Gatekeeper Code

Several concepts described in the earlier chapters are used in this demonstration:

- tasks
- event groups
- message queue
- task notification

The tasks used in this demonstration are:

- The gatekeeper task (gatekeeper_task()) starting in line 64 of Listing 14-1.
- User task 1 (usr_task1()) starting in line 246. This task polls the push buttons and lights the corresponding LED when pushed.
- User task 2 (usr_task2()) starting in line 274 for blinking LED3.

The second user task is just to prove that our gatekeeper task can handle multiple independent client tasks.

The client tasks become simple because the gatekeeper API and task is doing the heavy lifting. The blinking usr_task2() is just a simple delay loop:

```
0274: static void usr_task2(void *argp) {
0275:   bool state;
0276:   short rc;
0277:
0278:   for (;;) {
0279:     delay(500);
0280:     rc = pcf8574_get(LED3);
0281:     assert(rc != -1);
0282:     state = !(rc & 1);
0283:     pcf8574_put(LED3,state);
0284:   }
0285: }
```

Line 280 obtains the current LED state and inverts it in line 282. Then this is sent inverted to the LED3 before the next loop iteration (line 283).

The usr_task1() event loop is only slightly more complicated:

```
0247:   static const struct s_state {
0248:     uint8_t   button;
0249:     uint8_t   led;
0250:   } states[3] = {
0251:     { BUTTON0, LED0 },
0252:     { BUTTON1, LED1 },
0253:     { BUTTON2, LED2 }
0254:   };
...
0266:     for ( unsigned bx=0; bx<NB; ++bx ) {
0267:       rc = pcf8574_get(states[bx].button);
0268:       assert(rc != -1);
0269:       rc = pcf8574_put(states[bx].led,rc&1);
0270:       assert(rc != -1);
0271:     }
```

This uses the gatekeeper API to poll the push buttons in line 267, and then drives the corresponding LED based upon the association established in the states array.

Gatekeeper Initialization

Before examination of the gatekeeper task and the API functions, we must look at one part of the setup() initialization:

```
0289: void setup() {
0290:   int app_cpu = xPortGetCoreID();
0291:   BaseType_t rc;  // Return code
0292:
0293:   // Create Event Group for Gatekeeper.
0294:   // This must be created before any using
0295:   // tasks execute.
0296:   gatekeeper.grpevt = xEventGroupCreate();
0297:   assert(gatekeeper.grpevt);
```

For safety, the API functions must know when the gatekeeper task is up and operational. This is done through the event group that is created in line 296, within setup(). The handle is saved in the gatekeeper static structure storage:

```
0044: static struct s_gatekeeper {
0045:   EventGroupHandle_t  grpevt;   // Group Event handle(*)
0046:   QueueHandle_t       queue;    // Request queue handle
0047:   uint8_t             states[N_DEV]; // Current states
0048: } gatekeeper = {
0049:   nullptr, nullptr, { 0xFF, 0xFF }
0050: };
```

Except for the API functions, these values are private to the gatekeeper task. The C++ minded individual could convert this into a first-class C++ object to guarantee private access. The temptation to use C++ was resisted here to keep this as straight forward as possible.

The gatekeeper task will create the message queue and save the handle in structure member gatekeeper.queue (this happens in line 74). This is performed before the event group is given the ready notification. The I/O state of both PCF8574P chips is maintained in gatekeeper.states array, although it could be kept within the task – there is no external communication requirement for it.

Gatekeeper API Functions

The gatekeeper API functions hide the mechanism used for communication. The message queue gatekeeper.queue is used to ship the caller's request to the gatekeeper task. Line 172 is used to range check the XGPIO number (also known as a port in the code):

```
0167: static short pcf8574_get(uint8_t port) {
0168:   s_ioport ioport;      // Port pin (0..15)
0169:   uint32_t notify;      // Returned notification word
0170:   BaseType_t rc;
```

```
0171:
0172:   assert(port < 16);
```

Then the structure s_ioport, defined in line 168 is populated. Let's examine its declaration:

```
0053: struct s_ioport {
0054:   uint8_t input : 1;  // 1=input else output
0055:   uint8_t value : 1;  // Bit value
0056:   uint8_t error : 1;  // Error bit, when used
0057:   uint8_t port : 5;   // Port number
0058:   TaskHandle_t htask; // Reply Task handle
0059: };
```

Students may be unfamiliar with bit fields that can be used within a structure. Here the value of struct member input is one bit in size. When true it indicates that the request is a GPIO *get* request. When it is false, the member value (1 bit in size) holds the bit value to be written to the GPIO port. Member error is also declared here because, in the early stages of program development, the error was returned in the same structure. The required port number (XGPIO) is provided in the member port, which is 5 bits in size. This member can support a XGPIO number from 0 to 31. Finally, member htask provides the requesting task handle. This will be important later in the API processing function.

For the get request, the following setup is important:

```
0173:   ioport.input = true;  // Read request
0174:   ioport.port = port;   // 0..15 port pin
0175:   ioport.htask = xTaskGetCurrentTaskHandle();
```

The unused elements of the structure are left uninitialized. In safety-critical applications, they should be set to default values (a good practice). The next call checks to make certain that the gatekeeper task is ready:

```
0177:   pcf8574_wait_ready(); // Block until ready
```

This is managed by the function:

```
0150: static void pcf8574_wait_ready() {
0151:
0152:   xEventGroupWaitBits(
0153:     gatekeeper.grpevt, // Event group handle
0154:     GATEKRDY,     // Bits to wait for
0155:     0,            // Bits to clear
0156:     pdFALSE,      // Wait for all bits
0157:     portMAX_DELAY // Wait forever
0158:   );
0159: }
```

It simply calls the xEventGroupWaitBits() function, which will block until the GATERDY bit is set (set by the gatekeeper task when it becomes ready).

One the API knows that the gatekeeper is ready, it can send its request using the supplied gatekeeper.queue handle:

```
rc = xQueueSendToBack(
  gatekeeper.queue,
  &ioport,
  portMAX_DELAY);
assert(rc == pdPASS);
```

This ships the request to the gatekeeper, to be processed.

```
// Wait to be notified:
rc = xTaskNotifyWait(
  0, // no clear on entry
  IO_RDY|IO_ERROR|IO_BIT, // clear on exit
  &notify,
  portMAX_DELAY);
assert(rc == pdTRUE);
```

Here we see the purpose of providing the task handle to the gatekeeper task in the s_ioport message. The API call will block at this point until the gatekeeper task notifies that it has processed the request. Three bit values are monitored by this task notification:

- **IO_BIT** – used by get operations to return the data bit value.
- **IO_ERROR** – used by the gatekeeper to indicate if the operation was successful or not.
- **IO_RDY** – when true, it indicates that the operation has completed.

All three bits are necessary as part of the notification because if the IO_BIT is a 0-bit, and there is no IO_ERROR (it will be a 0-bit), then we must have at least the IO_RDY bit set for the notification to proceed. The IO_RDY bit is the primary driver here and the other bits are supplementary information. Receiving the notification this way saves us from having to provide a reply queue. Each client task would need one of its own unless some other arrangement was provided.

At the end of the API function, we can finally return the success or fail indication:

```
return (notify & IO_ERROR)
  ? -1
  : !!(notify & IO_BIT);
```

If the IO_ERROR bit is true, then the function returns -1 to indicate that the I2C request failed. Otherwise, a 1-bit or 0-bit is returned to provide the fetched GPIO value. The !! operator forces the evaluation to return a 1 or 0 (otherwise a 0b0010 & IO_BIT expression might return 0b0010 instead).

The pcf8574_put() is almost identical except that it must provide the value to be written to the port (line 211):

```
0210:   ioport.input = false; // Write request
0211:   ioport.value = value; // Bit value
0212:   ioport.port = port;   // PCF8574 port pin
0213:   ioport.htask = xTaskGetCurrentTaskHandle();
```

Gatekeeper Task
The gatekeeper task is almost your typical server task. It performs the following basic functions:

1. Initializes (lines 73 to 88). Creates message queue, starts I2C support and initializes the PCF8574P chips.
2. Announces that it is ready (line 91) by setting the bit GATERDY bit.
3. Event loop (lines 94 to 146).

The event loop consists of the following basic steps:

- Receives the request (lines 98 to 100).
- Converts the ioport.port (XGPIO) value into a devx (device index) and portx (port index). The device index is range checked (line 104), and the portx value is in the range 0 to 7 for the chip's pins P0 to P7 respectively.
- The I2C address is looked up and stored in variable addr (line 105).
- Then the request processing occurs for input (lines 107 to 119) or output (lines 121 to 131).
- The notify value, has the IO_RDY bit set (line 134), the error bit IO_ERROR when the operation fails, and the data value IO_BIT if the value was a 1-bit.
- The task is then notified of the operation's completion (lines 141 to 145). The task handle is tested in line 141, so that the task notification could be optional (this feature was not used in this demonstration).

Input from PCF8574P
When the operation requires that the PCF8574P XGPIO is read, the following steps occur:

1. The I2C device is provided the address and a read request (line 109).
2. When the read is successful, line 112 receives the data byte in the variable named data.
3. The request is marked successful (line 113).
4. The read value is stored in bit ioport.value for return (line 114).

If the I2C read operation fails for any reason, the following occurs:

1. The error flag is set in ioport.error (line 117).
2. The ioport.value is initialized to a 0-bit (line 118).

Output to the PCF8574P
When sending a data bit to the selected PCF8574P chip, the following takes place:

1. The last known data byte value for the chip is copied to variable data (line 122).
2. If the value to be written is a 1-bit, this is then ORed into data (line 124). Otherwise, the corresponding bit is set to zero (line 126);
3. The I2C is notified of a write operation with the selected I2C address (line 127).
4. The 8-bits of port data is written out (line 128).
5. The success of the transaction is tested (line 130). If the operation is successful, the new data byte is saved in the gatekeeper.states array (line 131).

PCF8574P State Management
The alert reader will notice that the gatekeeper does not update the gatekeeper.states array when *reading* the XGPIO pins. There is an important and subtle point there. The PCF8574P is a quasi-bidirectional device. To read an input, you must write a 1-bit to the port first. This in effect acts as a pull-up on the pin. The driving signal then pulls the pin down to ground potential so that it will read as a 0-bit, or left pulled-up to read as a 1-bit. The PCF8574P pull-up strength is no more than 100 μA, so there is no problem in this arrangement.

If the gatekeeper was to record the fact that it read a 0-bit *input* into the gatekeeper.states array, then a future output for an LED will also write a 0-bit out for that input pin! Once that has happened, it is no longer possible to read a 1-bit on the corresponding *input*. Let's clarify this with a concrete example:

1. Assume we have saved 0b11111111 for the first PCF8574P, where our push button 2 (PB2) is connected and all pins are in the high state.
2. When push button 2 is pressed, it grounds pin P4.
3. This causes 0b11101111 to be read (P4 is a 0-bit).
4. If this value were to be saved in gatekeeper.states[0], its value will affect future outputs. Assume for now that this has been erroneously coded. Note, no change to the PCF8574P has occurred yet apart from sensing this level change.
5. Because PB2 was sensed, the usr_task1 task outputs a 0-bit to LED2 to light it. This causes the value 0b11100111 to be written out and saved to gatekeeper. states[0].
6. Note that port P4 (for button 2) has been written out as a 0-bit.
7. When button 2 is released, the port P4 remains as a 0-bit (oops).
8. Because P4 has been written out as a 0-bit, the input pin will not pull-up to a 1-bit with the button release.

Managing the state of devices can be tricky.

Troubleshooting

There are quite a few wiring connections involved in breadboarding this particular demonstration resulting in increased potential for failure. For this reason, I've listed some tips for getting the demonstration up and running. The demonstration should immediately start blinking the LED attached to P0 (pin 4) of the second PCF8574P chip after receiving power and time enough to reset. There is no need for the Serial Monitor to be connected. If you fail to see LED3 blinking, then try the following:

1. Check the chip orientation of the PCF8574P chips on the breadboard. Are the oriented correctly so that you have the $+V_{CC}$ (pin 16) and Ground (pin 8) connections correctly identified and powered?
2. Check the grounding of both of the PCF8573P chips. Pin 8 of both chips must share a ground connection with the ESP32 and the power source. The power source is usually from the ESP32 dev board's +3.3 volt pin, derived from the USB cable connection.
3. Check with a multimeter that the pin 16 of both chips are receiving +3.3 volts. Make certain this is not +5 volts, since this may cause damage to the ESP32 chip.
4. Check that the connections for SDA (ESP GPIO 25) and SCL (ESP GPIO 26) are correct. Are these wires correctly connected to the PCF8574P pins 15 (SDA) and 14 (SCL)? For both chips?
5. Check that the address connections for A0, A1, and A2 are wired. All of these connect to ground except for A0 for the second chip.
6. Check the polarity of the LEDs used (with a series 220-ohm resistor). Confirm the operation of the LED by applying +3.3 volts to the LED and resistor pair. If that fails, try reversing the polarity of the LED resistor pair.
7. If still no success, try a lower V_F LED. Use a red LED if possible, since they have the lowest voltage drop (usually about 1.63 volts).
8. Check with the Serial Monitor connected – is there an abort reported?

```
assertion "!rc" failed: file ".../gatekeeper/gatekeeper.ino", line 87,
function: void gatekeeper_task(void*)
abort() was called at PC 0x400d9087 on core 1
```

This fail occurs at the I2C transmission (line 87) to one of the PCF8574P chips. Check all of the above to correct a possible I2C problem. Also, try:

1. Wiggling wires. All it takes is one bad connection to bring about a fail. Sometimes a Dupont wire will fail.
2. If you have spares, carefully replace the PCF8574P chips, observing ESD protection. If you lack an anti-static (ESD) wrist band, try keeping yourself grounded while performing the procedure. You may want to replace one at a time to locate the cause of the problem. Do put the cat out of the room while doing this (and ground yourself to unload the static).

If you have the LED3 blinking, but the buttons are unresponsive, check the following:

1. Measure the voltage at pins 9 and 10 of the first chip (button inputs) and pin 9 of the second chip (LED outputs for each button). Without buttons pressed, you should read +3.3 volts. Pressing the associated button should reduce this to 0 volts.
2. If the prior test passes, then the fault must lie with the associated LEDs. First measure the output of the LED output pin (pin 7 for PB0, pin 6 for PB1, pin 7 for PB2). The outputs should measure high with no button pressed but otherwise, go low when its button is pushed. If this is happening, then the LED and resistor pair is suspect or has incorrect polarity.

Sometimes a chip can "half work". It may seem ok, but fail to fully deliver or behave sporadically. I once spent a couple of hours assuming the code was at fault when in the end it was determined to be the PCF8574P chip. Check for this early to avoid early male pattern baldness. When ordering these chips, do get extras. These chips are economical enough to allow for extras. Observe ESD preventative measures.

```
// gatekeeper.ino

// GPIOs used for I2C
#define I2C_SDA       25
#define I2C_SCL       26

// Set to 1 for PCF8574A
#define PCF8574A      0

// Buttons:
#define BUTTON0       12    // P4 on dev1
#define BUTTON1       5     // P5 on dev0
#define BUTTON2       4     // P4 on dev0
#define NB            3     // # Buttons

// LEDs:
#define LED0          1     // P1 on dev0
#define LED1          2     // P2 on dev0
#define LED2          3     // P3 on dev0

#define LED3          8     // P0 on dev1

#include <Wire.h>

#if PCF8574A
// Newer PCF8574A addresses
#define DEV0          0x38
#define DEV1          0x39
```

```
#else
// Original PCF8574 addresses
#define DEV0          0x20
#define DEV1          0x21
#endif

#define N_DEV         2         // PCF8574 devices
#define GATEKRDY      0b0001    // Gatekeeper ready

#define IO_RDY        0b0001    // Task notification
#define IO_ERROR      0b0010    // Task notification
#define IO_BIT        0b0100    // Task notification

#define STOP          int(1)    // Arduino I2C API

static struct s_gatekeeper {
  EventGroupHandle_t  grpevt;   // Group Event handle(*)
  QueueHandle_t       queue;    // Request queue handle
  uint8_t             states[N_DEV]; // Current states
} gatekeeper = {
  nullptr, nullptr, { 0xFF, 0xFF }
};

// Message struct for Message/Response queue
struct s_ioport {
  uint8_t input : 1;  // 1=input else output
  uint8_t value : 1;  // Bit value
  uint8_t error : 1;  // Error bit, when used
  uint8_t port : 5;   // Port number
  TaskHandle_t htask; // Reply Task handle
};

// Gatekeeper task: Owns I2C bus operations and
// state management of the PCF8574P devices.

static void gatekeeper_task(void *arg) {
  static int i2caddr[N_DEV] = { DEV0, DEV1 };
  int addr;             // Device I2C address
  uint8_t devx, portx;  // Device index, bit index
  s_ioport ioport;      // Queue message pointer
  uint8_t data;         // Temp data byte
  uint32_t notify;      // Task Notification word
  BaseType_t rc;        // Return code

  // Create API communication queues
  gatekeeper.queue = xQueueCreate(8,sizeof ioport);
```

```
assert(gatekeeper.queue);

// Start I2C Bus Support:
Wire.begin(I2C_SDA,I2C_SCL);

// Configure all GPIOs as inputs
// by writing 0xFF
for ( devx=0; devx<N_DEV; ++devx ) {
  addr = i2caddr[devx];
  Wire.beginTransmission(addr);
  Wire.write(0xFF);
  rc = Wire.endTransmission();
  assert(!rc); // I2C Fail?
}

// Indicate gatekeeper ready for use:
xEventGroupSetBits(gatekeeper.grpevt,GATEKRDY);

// Event loop
for (;;) {
  notify = 0;

  // Receive command:
  rc = xQueueReceive(gatekeeper.queue,&ioport,
    portMAX_DELAY);
  assert(rc == pdPASS);

  devx = ioport.port / 8;   // device index
  portx = ioport.port % 8;  // pin index
  assert(devx < N_DEV);
  addr = i2caddr[devx];     // device address

  if ( ioport.input ) {
    // COMMAND: READ A GPIO BIT:
    Wire.requestFrom(addr,1,STOP);
    rc = Wire.available();
    if ( rc > 0 ) {
      data = Wire.read();   // Read all bits
      ioport.error = false; // Successful
      ioport.value = !!(data & (1 << portx));
    } else {
      // Return GPIO fail:
      ioport.error = true;
      ioport.value = false;
    }
  } else {
      // COMMAND: WRITE A GPIO BIT:
      data = gatekeeper.states[devx];
      if ( ioport.value )
        data |= 1 << portx;   // Set a bit
      else
        data &= ~(1 << portx); // Clear a bit
      Wire.beginTransmission(addr);
      Wire.write(data);
      ioport.error = Wire.endTransmission() != 0;
      if ( !ioport.error )
        gatekeeper.states[devx] = data;
    }

    notify = IO_RDY;
    if ( ioport.error )
      notify |= IO_ERROR;
    if ( ioport.value )
      notify |= IO_BIT;

    // Notify client about completion
    if ( ioport.htask )
      xTaskNotify(
        ioport.htask,
        notify,
        eSetValueWithOverwrite);
  }
}

// Block caller until gatekeeper ready:
static void pcf8574_wait_ready() {

  xEventGroupWaitBits(
    gatekeeper.grpevt, // Event group handle
    GATEKRDY,     // Bits to wait for
    0,            // Bits to clear
    pdFALSE,      // Wait for all bits
    portMAX_DELAY // Wait forever
  );
}

// Get GPIO pin status:
// RETURNS:
//  0  - GPIO is low
//  1  - GPIO is high
//  -1 - Failed to read GPIO

```

```
static short pcf8574_get(uint8_t port) {
  s_ioport ioport;      // Port pin (0..15)
  uint32_t notify;      // Returned notification word
  BaseType_t rc;

  assert(port < 16);
  ioport.input = true;  // Read request
  ioport.port = port;   // 0..15 port pin
  ioport.htask = xTaskGetCurrentTaskHandle();

  pcf8574_wait_ready(); // Block until ready

  rc = xQueueSendToBack(
    gatekeeper.queue,
    &ioport,
    portMAX_DELAY);
  assert(rc == pdPASS);

  // Wait to be notified:
  rc = xTaskNotifyWait(
    0, // no clear on entry
    IO_RDY|IO_ERROR|IO_BIT, // clear on exit
    &notify,
    portMAX_DELAY);
  assert(rc == pdTRUE);

  return (notify & IO_ERROR)
    ? -1
    : !!(notify & IO_BIT);
}

// Write GPIO pin for a PCF8574 port:
// RETURNS:
//  0 or 1: Succesful bit write
//  -1:     Failed GPIO write

static short pcf8574_put(uint8_t port,bool value) {
  s_ioport ioport;      // Port pin (0..15)
  BaseType_t rc;
  uint32_t notify;      // Returned notification word

  assert(port < 16);
  ioport.input = false; // Write request
  ioport.value = value; // Bit value
  ioport.port = port;   // PCF8574 port pin
  ioport.htask = xTaskGetCurrentTaskHandle();

  pcf8574_wait_ready(); // Block until ready

  rc = xQueueSendToBack(
    gatekeeper.queue,
    &ioport,
    portMAX_DELAY);
  assert(rc == pdPASS);

  // Wait to be notified:
  rc = xTaskNotifyWait(
    0, // no clear on entry
    IO_RDY|IO_ERROR|IO_BIT, // clear on exit
    &notify,
    portMAX_DELAY);
  assert(rc == pdTRUE);

  return (notify & IO_ERROR)
    ? -1
    : !!(notify & IO_BIT);
}

// User task: Uses gatekeeper task for
// reading/writing PCF8574 port pins.
//
// Pins:
//  0..7    Device 0 (address DEV0)
//  8..15   Device 1 (address DEV1)
//
// Detect button press, and then activate
// corresponding LED.

static void usr_task1(void *argp) {
  static const struct s_state {
    uint8_t   button;
    uint8_t   led;
  } states[3] = {
    { BUTTON0, LED0 },
    { BUTTON1, LED1 },
    { BUTTON2, LED2 }
  };
  short rc;

  // Initialize all LEDs high (inactive):
  for ( unsigned bx=0; bx<NB; ++bx ) {
    rc = pcf8574_put(states[bx].led,true);
    assert(rc != -1);
  }

  // Monitor push buttons:
  for (;;) {
    for ( unsigned bx=0; bx<NB; ++bx ) {
      rc = pcf8574_get(states[bx].button);
      assert(rc != -1);
      rc = pcf8574_put(states[bx].led,rc&1);
      assert(rc != -1);
    }
  }
}

static void usr_task2(void *argp) {
  bool state;
  short rc;

  for (;;) {
    delay(500);
    rc = pcf8574_get(LED3);
    assert(rc != -1);
    state = !(rc & 1);
    pcf8574_put(LED3,state);
  }
}

// Initialize Application

void setup() {
  int app_cpu = xPortGetCoreID();
  BaseType_t rc;  // Return code

  // Create Event Group for Gatekeeper.
  // This must be created before any using
  // tasks execute.
  gatekeeper.grpevt = xEventGroupCreate();
  assert(gatekeeper.grpevt);

  // Start the gatekeeper task
  rc = xTaskCreatePinnedToCore(
    gatekeeper_task,
    "gatekeeper", // Name
    2000,         // Stack size
    nullptr,      // Argument
    2,            // Priority
    nullptr,      // Handle ptr
    app_cpu       // CPU
  );
  assert(rc == pdPASS);

  // Start user task 1
  rc = xTaskCreatePinnedToCore(
    usr_task1,   // Function
    "usrtask1", // Name
    2000,       // Stack size
    nullptr,    // Argument
    1,          // Priority
    nullptr,    // Handle ptr
    app_cpu     // CPU
  );
  assert(rc == pdPASS);

  // Start user task 2
  rc = xTaskCreatePinnedToCore(
    usr_task2,  // Function
    "usrtask2", // Name
    2000,       // Stack size
    nullptr,    // Argument
    1,          // Priority
    nullptr,    // Handle ptr
    app_cpu     // CPU
  );
  assert(rc == pdPASS);
}

// Not used:
void loop() {
  vTaskDelete(nullptr);
}
```

Listing 14-1. Demonstration gatekeeper task program gatekeeper.ino.

Summary

The demonstration provided in this chapter highlights the utility of the gatekeeper task. The gatekeeper API makes the user tasks trivial to write. All of the complexity of I2C, chip and state management was performed within the gatekeeper task and its interfacing functions. In larger applications, this type of component decoupling can lead to time savings in development and testing. This also enhances the application security, which has come under increased scrutiny in these times.

Exercises

1. What is the danger of accessing the gatekeeper before it is ready (in other words, why was the event group used in the demonstration program)?
2. How does the gatekeeper API simplify the application code?
3. How does the gatekeeper approach ease testing?
4. How does the gatekeeper approach enhance the security of the application?
5. What is one resource disadvantage of the gatekeeper task approach?

Chapter 15 • Tips and Hints

The show has come to an end.

Everyone starts somewhere, including the experts. This chapter contains some tips and hints for beginning Arduino programmers. Arduino was developed for students wading into the embedded programming scene, allowing them to get their feet wet without being overwhelmed with technical detail. Because some Arduino enthusiasts and hobbyists may lack formal software training, there are a few things worth mentioning "by the way" that most seasoned practitioners take for granted.

Forums: Invest Some Effort

Forums frequently contain posts asking for help to develop something complex. The request is often in the form of "I want to do ... I am a newbie- can somebody help?" It's not a crime to be unaware of the complexity and it's also ok to be a newbie. But how long did it take to type this request? Ten seconds? How much effort will be made by the responders? Maybe ten seconds if you get a response at all.

People are much more willing to help if they see that you've invested some effort on your own. Have you laid out what you think is needed and how you think you're going to get there? It's ok to be wrong about that because it demonstrates that you aren't just throwing the problem over the wall and hoping someone else will do all the work.

Start Small

Sometimes requests are made to help program large and complicated assignments. The usual response to these posts is no reply at all. It's not that no one knows the answers to the query but that nobody wants to expend the effort to tutor the user on the many things that need to be learned first. If the user must continue with that assignment, break it up into smaller parts. Then ask specific questions about the difficulties that are challenging.
The best approach, however, is to start small with simpler assignments. Everyone knows this but enthusiasts get impatient. Are you the type of person that just wants "the answer" or do you want to know how to determine the answer? Experience is the best teacher.

There's a reason that people start with blinking LEDs. LEDs are simple- they turn on or turn off. As simple as that is, there are still lessons to be learned. For example, if the LED does not light, what is the cause? If you've never encountered that before, you might not realize that the LED was wired with the wrong polarity. Don't cheat yourself out of these training moments.

The Government Contract Approach

New programmers, armed with the knowledge of their programming language, often leap into writing the whole program at once and then try to debug it. I like to refer to this as the government contract approach. Once armed with requirements, the software is written up and then tested at the end. The ensuing debugging sessions can create a level of insanity. Don’t get me wrong – requirements are important. When the requirements are coming from ourselves (in the hobby), they change frequently. Or if you’re developing something just for fun, you might not even have firm requirements. These are just some reasons to avoid coding *everything* before testing.

The best reason to avoid the government contract approach is that debugging is more difficult on embedded devices. There may be no debugger available or its ability to trace is limited. You can’t step through an interrupt service routine, for example. A better approach is to start small and use the basic shell and stub approaches.

The Basic Shell

Rather than attempt to write the entire application before debugging it, write a basic shell first. In this shell of a program, code a minimal setup() and loop() function for your Arduino code. Especially when using a dev board ESP32, take advantage of the USB to serial link using the Serial Monitor. This will simplify your development and test cycle.

The first shell might just be a "Hello from setup()" and a "Hello from loop()" in the loop() function, as illustrated in Listing 15-1. Remember that this first program need not be elegant. By trying this much at the beginning, you prove a full compile, flash upload, and run test cycle. The delay() call in line 4 allows the ESP32 libraries to establish the USB serial link before running on ahead. Part of the proof of concept is to just establish proof of a debugging serial link with the Serial Monitor.

```
0001: // basicshell.ino
0002:
0003: void setup() {
0004:   delay(2000); // Allow for serial setup
0005:   printf("Hello from setup()\n");
0006: }
0007:
0008: void loop() {
0009:   printf("Hello from loop()\n");
0010:   delay(1000);
0011: }
```

Listing 15-1. A sample basic starting shell of a program.

The Stub Approach

Obviously, you want your application to do more than that basic shell. Start building on that framework by adding stub functions (Listing 15-2). Functions init_oled() and init_gpio() are just stubs for what will become initialization functions for the OLED and the GPIO devices.

Compile, flash, and test what you have. Did it run? The output from Listing 15-2 should be like the following in the Serial Monitor:

```
Hello from setup()
init_oled() called.
init_gpio() called.
Hello from loop()
Hello from loop()
Hello from loop()
...
```

The next step is to expand upon what the stub functions do – the actual device initialization. Avoid the temptation to do it all and keep the code additions small and incremental. This will save serious head-scratching when new problems develop. It's amazing how even small obvious additions can bring about so much trouble.

```
// stubs.ino

static void init_oled() {
  printf("init_oled() called.\n");
}

static void init_gpio() {
  printf("init_gpio() called.\n");
}

void setup() {
  delay(2000); // Allow for serial setup
  printf("Hello from setup()\n");
  init_oled();
  init_gpio();
}

void loop() {
  printf("Hello from loop()\n");
  delay(1000);
}
```

Listing 15-2. The basic shell program expanded with stubs.

Block Diagrams

Larger applications may benefit from a block diagram for planning out the FreeRTOS tasks needed. The setup() and loop() functions start from the "loopTask", which is provided by default. If you don't like the loopTask stack allocation you can delete that task by calling vTaskDelete(nullptr) from loop() or from within setup(). What additional tasks do you need? Will ISR routines be feeding events to any of them? Draw lines where there might be message queues. Perhaps use dotted lines where events, semaphores or mutexes are involved between tasks. It doesn't have to be a UML approved diagram – just use conventions that make sense to you.

As part of stubbing out your application, create each *task* initially as a stub function. All that function needs to do is to announce its start. A task is not permitted to return in FreeRTOS, so for stub purposes, the task can delete its task after the announcement. Later on, you can fill the task in with the final code.

Faults

As you build up your application one portion at a time, you may suddenly run into a program fault of one kind or another. This can be most vexing in a finished application. But because you are developing your application by adding one portion of code at a time, you already know what the added code was. The fault is likely to be related to the added code. The Arduino compiler options used don't allow the compiler to warn about everything that it should. Or it may be the way that the header file for the newlib printf() support is defined. Either way, an example of code that can lead to faults is this:

```
printf("The name of the task is ‘%s’\n");
```

See the problem? There should be a C string argument after the format string, to satisfy the "%s" format item. The printf() function will expect it and will reach into the stack to get it. But the value it finds may be garbage or nullptr and cause a fault. The compiler knows about these problems but these warnings are not reported for some reason.

Another common source of faults is running out of stack space. If you can't immediately locate the cause of the fault, allocate additional task stack space for all added tasks. This may eliminate the fault. Once you finish testing, the various stack allocations can be carefully reduced.

There is also the issue of object lifetimes to be aware of. Did you send a pointer through a queue? See the section Know Your Storage Lifetimes. Or was the C++ object destructed by the time another task tried to access it?

Know Your Storage Lifetimes

When you're starting out, there seems to be so much to learn. Don't let that discourage you but do examine the following code snippet:

```
static char area1[25];

void function foo() {
  char area2[25];
```

Where is the storage for array area1 created? Is it the same as area2? The array area1 is created in a region of SRAM permanently allocated to that array. That storage never goes away.

The storage for area2 is different however because it is allocated *on the stack*. The moment when function foo() returns, that storage is released. If you passed the pointer to area2 by a message queue, for example, that pointer will be invalid the moment that foo() returns.

If you want the array array2 to persist after foo() returns, you can declare it static within the function:

```
static char area1[25];

void function foo() {
  static char area2[25];
```

By adding the static keyword to area2's declaration, we have moved its storage allocation to the same region as area1 (i.e. *not* on the stack).

Note that the array array1 also is declared with the static attribute, but in that case, the static keyword has a different meaning (when declared outside of the function). Outside of a function, the static keyword just means don't assign an external symbol to it ("area1"). Declaring these variables with static avoids link step conflicts.

Avoid External Names
Functions and *global* storage items that are only referenced by your current source file should probably be declared static. Without the static keyword, the name becomes "extern" and might interfere with other linked in libraries. Unless your function or global needs to be extern, declare them as static.

The functions setup() and loop() on the other hand, must be extern symbols because the linker must call them from an application startup module. Being extern allows the linker to locate and link with them.

Leverage Scope
A software best practice to embrace is to limit the scope of entities so that they cannot be confused or referenced from places where they shouldn't be. Declaring everything globally is convenient for small projects but can become a headache for larger applications. I like to refer to this as the cowboy programming style. Programmers that remember COBOL, will relate.

The problem with the cowboy style is that if you find a bug where something is being used/ modified when it shouldn't be, it becomes difficult to isolate. When the language's scope rules are used instead, the compiler will tell you upfront when you're trying to access something that shouldn't be. Permitted access will be enforced.

One way to limit the reach of FreeRTOS handles and other data items is to pass them into the tasks as members of a structure. For example, if one task needs the handle to a queue and a mutex, then pass these items in a structure to the task at task creation time. Then, only the using task knows about these handles.

Rest the Brain
When it comes to the human experience, psychologists tell us that there are at least 16 different personality types (Myers-Briggs). But I believe that most people will continue to work on a problem after they have given up consciously thinking about it. So when you find yourself losing patience in a late-night debugging session, allow yourself some rest. It might also save you from taking unnecessary risks, which may end in the magic smoke.

Some may sleep soundly, while others will toss and turn in the night. But the mind mulls the day's events and runs through all of the possible scenarios. In the morning your spouse might complain about your hexadecimal mumblings in your sleep. But when you awake, you will usually have some new ideas to try. If it was a particularly difficult problem, the eureka moment may take a few days to develop. You will conquer.

Note Books
When your head hits the pillow at night, you might suddenly recall a thing or two that you forgot to attend to in the code or circuit. A notepad by the bedside might be a useful memory aid. When you're young, the mind is uncluttered, and remembering things is easy. As you age, however, life becomes filled with complications and then things will start to fall through the cracks. A notebook is useful to record what was tried or how you solved an issue. Monday mornings at work benefit from being able to pick up where you left off from the previous Friday.

Unless you use a particular technique frequently, you will need to look it up. This is another way that note-keeping is helpful. Make notes of those special APIs, C++ techniques, or FreeRTOS things that you found useful. If you prefer to be able to copy and paste, enlist the use of sites like evernote.com. These have the advantage of being electronically searchable.

Asking for Help
Since the emergence of the "wild woolly interwebs", we have the blessing of forums and search engines, which allow us to "goggle" for help. A web search is often a fruitful first step for immediate answers or clues. But take what you read with a grain of salt – not all advice is good. Depending on the nature of the problem, you'll often discover that others have experienced similar problems. In that case, you may get one or more posted answers to work with.

When reaching out to a forum, ask intelligent questions. Forum posts like "My I2C doesn't work, can you help?" shows very little initiative. This is another "throw the problem over the wall and hope for the best" type of effort. Would you take your car to a garage and just tell them that your car is broken? Forum posts shouldn't need to play the twenty questions game.

Post your query with some *specific* information:

- The precise nature of the problem (what part of I2C is not "working")
- What I2C devices are you working with?
- What have you tried so far?

- Perhaps the specifics of your ESP32 dev board.
- Development platform – Arduino or ESP-IDF?
- What libraries, if any, are you using?
- Any other observable strangeness.

I would avoid posting code on the first post but use your best judgement. Some people post oodles of code as if this makes it self explanatory. I believe it is more productive to explain the nature of the problem first. You can always post the code as a followup.

When posting code, it may not always be necessary to post all of it (especially when lengthy). Sometimes it is only necessary to post what is likely contributing to the problem. In our example, you might only post the I2C code functions used.

Forums usually have a way to post "code" in the message (like [code] ... [/code]). Be sure to make use of that whenever possible. Otherwise, between the proportional font and the lack of respect for indentation, the code becomes a horrible mess to read. I hate reading dreadfully indented code.

There is a beneficial side-effect to precisely describing the problem, whether in a post or by email – by the time you finish describing the problem, you might already realize the answer. Alternatively, when working with a colleague or fellow student, just explaining the problem to them can produce the same result.

Divide and Conquer
The new student can be challenged by an application that seizes up. How do you isolate the section of code responsible? The seasoned programmer knows the divide and conquer technique.

The concept is as simple as the guess the number game. If you must guess a number that I am thinking of between 1 and 10, and you guess 6 and I answer that the number is lower, then you'll divide that again with a guess of perhaps 3. Eventually, you'll be able to guess the number by reducing the ranges with each try. When a program seizes up, you divide it into halves until you isolate the region of offending code.

Different methods can be used for the indicator- a print to the Serial Monitor or the activation of an LED. The LED is useful for ISR tracing where you can't print messages. If you have enough GPIOs, you can even use a bicolour LED to signal different things. The idea is to indicate that the code was executed at the points of interest. If you need more from your LED, you can blink codes when not in an ISR. Once you narrowed down the region of code where the problem occurs, you can scrutinize the code more carefully for the cause.

Programming for Answers
I have seen programmers in the workplace argue for half an hour over what happens when such and such occurs. Even after that, the argument often remains unresolved. The whole issue can often be settled by writing a simple one minute program to test the hypothesis. Of course, use some common sense with what you've observed:

- Is the observed behaviour supported by the API?
- Or is this behaviour due to misusing the API or exploiting a bug?

If the API is open-sourced, then the source code is often the final answer. Often the code and comments will indicate the intention of poorly documented interfaces. Conclusion: don't be afraid to write throw-away code.

Leverage the find Command
When checking open-sourced code, you can look for it online or examine what you have installed on your system. Looking at installed code is important when you think you've found a bug in a library that you're using. One of the disadvantages of Arduino is that a lot of things are performed behind the scenes and remain hidden to the student. If you're using a POSIX system (Linux, FreeBSD or MacOS etc.) then the find command is extremely useful. Windows users can install WSL (Windows Subsystem for Linux) to accomplish the same or use a windows version of the command.

Make time to make friends with the find command. It is powerful and looks daunting to the newbie, but there is no great mystery there. Just a lot of flexibility that can be absorbed in small waves. The find command supports a myriad of options that make it *seem* complicated. Let's examine the most important and useful of these. The basic command format follows the following general form:

```
find [options] path1 path2 ... [expression]
```

The options in the command line are for more advanced users, and we can safely ignore them here. The one or more pathnames are the directory names where you want the search to begin. To get results, it used to be necessary to specify the -print option for the expression component but with the Gnu find command, this is now assumed by default:

```
$ find basicshell stubs -print
```

or just:

```
$ find basicshell stubs
basicshell
basicshell/basicshell.ino
stubs
stubs/stubs.ino
```

With the above form of the command, all pathnames descending from the given directory names are listed. This includes directories and files. Let's restrict the output to files with the type option with the argument "f" (indicating files):

```
$ find basicshell stubs -type f
basicshell/basicshell.ino
stubs/stubs.ino
```

Now the output shows only file pathnames. This output is still not that useful. What we need to do is to tell the find command to do something with these file names. The grep command is a great candidate:

```
$ find basicshell stubs -type f -exec grep ‘setup’ {} \;
void setup() {
  delay(2000); // Allow for serial setup
  printf("Hello from setup()\n");
void setup() {
  delay(2000); // Allow for serial setup
  printf("Hello from setup()\n");
```

We’re almost there, but first, let’s explain a couple of things. We added the find option exec followed by the name of the command (grep) and some special syntax. The argument ‘setup’ is the grep regular expression that we’re searching for (or a simple string). It will often need to be placed in single quotes to prevent the shell from messing with it. The "{}" argument indicates where on the command line to pass the pathname (to grep). Finally the "\;" token marks the end of the command. The latter is necessary because you might add even more find options after the provided command.

To be even more useful, we need to see the file name where grep found a match. In some cases you might also want the line number where the match was found. Both of these are satisfied by grep by using the H option (show pathnames) and n (show line numbers):

```
$ find basicshell stubs -type f -exec grep -Hn setup {} \;
basicshell/basicshell.ino:3:void setup() {
basicshell/basicshell.ino:4:  delay(2000); // Allow for serial setup
basicshell/basicshell.ino:5:  printf("Hello from setup()\n");
stubs/stubs.ino:11:void setup() {
stubs/stubs.ino:12:  delay(2000); // Allow for serial setup
stubs/stubs.ino:13:  printf("Hello from setup()\n");
```

This now provides all the detail that you could want.

Sometimes, we just want to find out where the header file is installed. In this case, we don’t want to grep the file, but only to locate where the file is. For example, where is the header file for the Arduino nRF24L01 library installed? The header file is named RF24.h. Lumen could use the following find command on her iMac:

```
$ find ~ -type f 2>/dev/null | grep ‘RF24.h’
/Users/lumen/Documents/Arduino/libraries/RF24/RF24.h
/Users/lumen/Downloads/RF24-1.3.4/RF24.h
```

The tilda (~) represents the home directory (in most shells). You could also specify $HOME or the directory explicitly. The clause "2>/dev/null" is added only to suppress error messages about directories that she lacks permission on (this happens a lot on the Mac). This is particularly useful if you end up searching through everything (starting with the root directory "/"). Allow lots of time when searching from root (definitely a coffee making moment). The find output in the earlier example is piped into grep so that it will only report those pathnames with the string RF24.h in them. This search can also be done using the find command's name option:

```
$ find ~ -type f -name ‘RF24.h’ 2>/dev/null
/Users/lumen/Documents/Arduino/libraries/RF24/RF24.h
/Users/lumen/Downloads/RF24-1.3.4/RF24.h
```

Either way, it is evident that on Lumen's iMac, the directory ~/Documents/Arduino/libraries/RF24 holds the header file RF24.h (and related files).

The name option also accepts file globbing searches. To search for all header files, you might try:

```
$ find ~ -type f -name ‘*.h’ 2>/dev/null
```

This just scratches the surface – it provides a tool too powerful to be ignored. Leverage it.

Infinitely Malleable

Some programmers will only develop their software until it "seems to work". Once they see the results they expected to see, they assume their job is done. Immediately washing their hands of it, they place it into production only to have it return for bug fixes. Sometimes repeatedly so.

For hobby work, you might protest that this is acceptable. Yet the same hobbyists will share their code. Do you want to inflict bad or embarrassing code on others? Set a bad example? Unlike a soldered electronic circuit, software is infinitely malleable, so don't be afraid to improve it. Software often needs polishing.

Ask yourself:

- Is there a better way this application could have been written?
    - Replace macro procedures with inline functions?
    - Use C++ templates, for generic code?
    - Would the use of the Standard Template Library (STL) improve the application?
    - Is there a more efficient way to perform some of the operations?
- Is the code easily understood?
- Easily maintained, or extended?
- Is the application vulnerable to exploits or misuse? Bug-free?
- Does the code make good use of the C/C++ scoping rules to restrict access to handles and other resources within the program?
- Are there memory leaks?
- Are there race conditions?
- Is there memory corruption under some conditions?

Take pride in your work and make it the best it can be. Properly done, it might serve you someday in a job application. Engineers are always looking to perfect their craft.

Make Friends with Bits

I often wince at occurrences of macros like BIT(x), designed to set a specific bit within a word. As a programmer, I favour to see the actual expression (1 << x) than I would a macro BIT(x). Using a macro requires trust that it is implemented the way that you assume it is. I hate assuming. Yes, I can look up the macro definition, but life is short. Is setting a bit so difficult that this indirection is necessary? I encourage all beginners to master bit manipulation in C/C++. My only caution is to be mindful of the order of operations. But this is easily fixed with some brackets around the expression.

This leads to taking the time to learn the precedence of C operators and the difference between & and &&. If you're unsure about these then invest in yourself. Learn them cold so that you can apply them for the rest of your life. I began my career with a small chart taped to my monitor. It was immensely helpful.

Efficiency

It seems that almost all new programmers start off obsessed with efficiency. To them, it is a badge of honour to code the most efficient version of an assignment. Don't get me wrong – there is a place for efficiency, like in an MPU where it must handle video encoding and decoding, with barely enough resources to do it. In general, however, the need is not as great as you might think.

A junior Linux programmer once complained to me about how inefficient a MySQL query was that he working within a C++ program. He had already spent more time puzzling over how to reduce this overhead than he would ever save in the code's optimization. The query ran once, or perhaps a few times per day at most. In the big picture, the efficiency of this component was irrelevant.

When you embark on a change for efficiency's sake, ask yourself if it matters in the big picture. Will the end-user notice the difference? Will it make the application code less easy to understand and maintain? Will the code be more secure? There was a time when computer time was valuable. In today's world, the cost of the human programmer is where the cost resides. If all you end up accomplishing is more time for the FreeRTOS idle task, then what have you accomplished?

Source Code Beauty

When I was young and full of fire in my belly, I received from my professor my first marked assignment for that semester, with less than full marks. I was quite offended because the program worked perfectly. So what was the issue? The problem was that it wasn't beautiful enough.

I forget the beauty specifics now, but the lesson stayed with me. You could say that the lesson scarred me in a good way. When I initially protested, he made the reply to the class that code is only written once but read many times. If the code is ugly or messy, it can be difficult to maintain and few people will want the task of maintaining it. He encouraged us to make the code easy to read and a thing of beauty. This includes nicely formatted code, nicely formatted comments, but not too many comments. Too much comment can obscure the code and get neglected in program maintenance.

Fritzing vs Schematics

I believe that working from Fritzing diagrams is a bad practice. A person wanting to become a painter does not continue with paint-by-number canvases. Yet this is exactly what Fritzing diagrams are. Like the paint-by-number hobbyist painter, it may be suitable for some that just want to *reproduce* the build. It, however, should be avoided by those looking forward to a career in the field.

Schematic diagrams, on the other hand, are visual representations of what the circuit is. They provide at a glance what a wiring diagram cannot. Can you absorb a circuit by looking at a mess of wires? I encourage the enthusiast to take the time to learn schematic symbols and conventions. Learn how to wire your projects from a schematic rather than a wiring diagram.

Pay now or Pay Later

Here is some general advice for students anticipating careers in programming, whether it be for embedded computing or otherwise. In healthy career development, you will start with junior assignments and move on to more senior assignments as your talent grows. Allow time and experience to grow that talent. Don’t get too ambitious and rush it.

There used to be an old North American Midas Muffler commercial in the 1970s along the lines of "you can pay me now or pay me later". The message was about early maintenance. Your career likewise needs early maintenance. If you work at it now, it will pay dividends for your career later. Don’t be afraid to put in some time and sacrifice in those early years.

Indispensable Programmers

My last piece of advice is related to employee attitude. After the early career years pass, a few programmers morph into a "job security" mode. They will build systems that are difficult to follow and otherwise keep information to themselves. They don’t like to share with other employees. The motivation for this is to become indispensable to the company. You don’t want to become an indispensable employee. Coworkers will dislike you and management will not tolerate it forever. They will take the hit if necessary to break that dependency. Companies don’t like to be held hostage.

There is another reason to shun indispensable – you’ll want to move onto new challenges and leave your old job functions behind (to a junior). Management will not assign you new and exciting challenges if you are needed to support those old functions. If that old stuff is too difficult to hand off to a junior, then that junior just might get that new opportunity instead of you. In the workplace, you want to be ready to take on new challenges.

Final Curtain

We’ve now reached the final curtain – the end of this book. But this is not the end for you because you’re going to take what you’ve practiced and apply those FreeRTOS concepts in applications of your own. I hope you’ve enjoyed the journey. Thank you for allowing me to be your guide.

Appendix A

Chapter 2 – Tasks

1. Use function xTaskGetCurrentTaHandle() to obtain the current task's handle.
2. You give up the CPU by calling taskYIELD(). If there are no other ready to execute tasks, the control will return to the caller.
3. The application CPU is CPU 1 on the dual-core ESP32. On single-core ESP32's, there is only CPU 0.
4. The default Arduino task name is the loopTask.
5. One task is suspended by calling vTaskSuspend().
6. A task can delete itself by calling vTaskDelete(), either with a null handle (indicating self) or with its specific task handle.
7. A task that deletes itself defers the release of its stack until the IDLE task can process the request.
8. The preemption occurs with the system tick interrupt.
9. The ESP32 tick interrupts occur 1 millisecond apart.

Chapter 3 – Queues

1. No, because an ISR has special requirements, which are met with FreeRTOS functions ending in the suffix "FromISR".
2. The function xQueuePeek() is used to peek at the next item in a queue, without removing it.
3. To indicate no timeout, supply the macro value portMAX_DELAY to the FreeRTOS function.
4. The size of the queue item is specified in the xQueueCreate() call.
5. When receiving a data item from a queue, the receiving item storage must be at least the same size or larger, than the size specified at queue create time.
6. To receive a data item immediately without blocking the caller's execution, specify a timeout value of 0 ticks, then check for success.
7. xQueueReceive() returns the value pdQUEUE_EMPTY when no data item was received.
8. xQueueReceive() always returns the data item at the front of the queue.

Chapter 4 – Timers

1. No, a timer callback must never make a blocking call like vTaskDelay().
2. A timer is created in the dormant state.
3. The timer callback uses the RTOS daemon task's (aka timer server task) stack.
4. Some timer functions make use of the timer command queue.
5. The FreeRTOS entity "Timer ID value" is essentially a user data pointer.
6. A pointer can be safely abused to carry an integer if the integer value's size is no larger than the pointer itself (32-bits on the ESP32).

Chapter 5 – Semaphores

1. One task blocks after a "take" operation and the other tasks unblocks it by performing a "give" operation.
2. There is no deadlock. If Task B also tries to take semaphore 1, or Task A also tries to take semaphore 2, a deadlock will occur.

3. Another commonly used name for a deadlock is the "deadly embrace".
4. A single binary semaphore can only unblock one task that is blocked in its take attempt.
5. A binary semaphore is always created empty (not "given").
6. A counting semaphore is created with a value matching the initial count, as provided in the xSemaphoreCreateCounting() function’s second argument.
7. Deadlock is prevented in the dining philosophers problem by limiting the maximum number of philosophers that will eat at one time. If there are four philosophers, then by limiting the maximum number wanting to eat to three, it is always possible for at least one to eat (grab both forks).
8. Giving a counting semaphore increases the count while taking reduces it.
9. When the counting semaphore is initialized to zero, all resources are marked as "taken". Giving the semaphore increases the count and makes a resource available.
10. The binary semaphore has two states: empty (not "given") and full ("given").

Chapter 6 – Mailboxes

1. Mailbox contents are fetched with xQueuePeek().
2. The mailbox is created empty, like a queue.
3. The xQueueOverwrite() allows overwriting the single queue entry if the mailbox (queue) is not empty.
4. The size of the mailbox data *item* is fixed at the time the mailbox (queue) is created. The size is carried within the mailbox.
5. The xQueueOverwrite() function does not block because if the mailbox (queue) is full, the data item is always overwritten.
6. The function xQueuePeek() will block if the mailbox (queue) is empty and the argument xTicksToWait is non-zero.
7. An empty mailbox can be useful to represent "no value".

Chapter 7 – Task Priorities

1. The most urgent priority for the ESP32 Arduino is 24, and the least urgent priority is 0.
2. You do *not* need to call vTaskStartSchduler() for the ESP32 Arduino, because the Arduino startup does this for you before calling the function setup().
3. The FreeRTOS scheduler is invoked at the system tick interrupt, and in a FreeRTOS API call that changes task priority, blocks or unblocks a task’s execution. Note: It can also be optionally invoked after servicing a non-tick interrupt (to be covered in a later chapter).
4. Tasks at equal priority are scheduled in a round-robin fashion (ignoring the ESP32 dual-core wrinkle).
5. A ready-to-go task is created by creating the task at a lower priority than current, suspend it, set its required priority, and then resume it when it is time for the task to start.
6. A task’s execution is pre-empted when another task has been made ready at a higher priority or has had its priority increased above the current task.
7. On the ESP32, the system tick interrupt occurs every 1 ms. Thus a full-time slice would be 1 ms in length.
8. A time slice that is not full occurs when another task has become blocked or yielded, giving control to another task before the next system tick interrupt.

9. The call to taskYIELD() directly invokes the FreeRTOS scheduler, but never blocks. It will schedule the next task in round-robin sequence. If there are no other tasks at the same priority, control returns immediately to resume the current task.
10. The vTaskDelay() function changes the task to the Blocked state until the specified number of ticks have elapsed.
11. Calling taskYIELD() invokes the FreeRTOS scheduler. If there are other tasks ready at the same priority, they will be scheduled round-robin.
12. A task that receives no CPU time is said to be CPU starved. A task becomes CPU starved when one or more higher priority tasks use all available CPU time. Only when a higher priority task is blocked or suspended, can a lower priority task get execution time.
13. The call to taskYIELD() never schedules a higher priority task because that task would already be running, if it existed in the ready state. The call to taskYIELD() will allow other equal priority tasks to schedule in round-robin fashion.

8 – Mutexes

1. A low priority task holding a shared binary semaphore can prevent the high priority task from running because it needs the same lock. The impact of this is that the high priority task may not run or allows lower priority task(s) to run instead. If the low priority task never gets scheduled, the high priority task may never run again.
2. A non-recursive mutex must *never* be locked (taken) more than once by the same task. If attempted, the task will hang indefinitely.
3. The mutex prevents priority inversion by temporarily boosting the low priority task to match that of the high priority task attempting to gain the lock. This permits the low priority lock holder to resume to the point of releasing the lock.
4. The boosted priority of the task changes back to the original priority the moment the mutex has been released (unlocked/given).
5. If a recursive mutex is not unlocked (given) the same number of times it has been locked (taken), the mutex remains locked.
6. It is unsafe to delete a mutex *or* a recursive mutex if there are task(s) blocked on them.
7. Yes. A mutex is initially unlocked (given) while the binary semaphore is initialized in the locked (taken) state.
8. The PCF8574 can only drive a LED by sinking current because it is unable to source more than about 300 uA. The device can sink about 25 mA, however.

9 – Interrupts

1. The function called from an ISR must not affect an interrupted call into the same function. To be callable, the function must be recursive and state free.
2. Routines like printf() and snprintf() often require considerable stack space. Additionally, on some platforms, the implementation may not be recursive.
3. The routines malloc() and free() often manage memory in lists of various structures. To call either from an ISR could scramble the list management since these are non-reentrant functions.
4. The ESP32 has a stack dedicated to interrupt processing. Its size is configured by the macro name CONFIG_FREERTOS_ISR_STACKSIZE, which is approximately 1536 bytes.
5. The stack space is reduced by each pending ISR call. Due to priority interrupts, nested calls to other ISR routines can occur before the present ISR completes.

6. There is no timeout parameter for the xQueueSendFromISR() function because an ISR function must never block its own execution.
7. The woken (third) argument to xQueueSendFromISR() allows the returned value to be set to pdTRUE, when a task of equal or higher priority has been awakened. When set to true, it indicates that the task scheduler should be invoked to allow the ISR to resume a different task upon return.
8. Macro portYIELD_FROM_ISR() invokes the task scheduler from an ISR.
9. If the task scheduler is not invoked from an ISR when the woken (third) argument to xQueueSendFromISR() returns true, a lower priority task may resume while a higher priority task is ready.
10. The FreeRTOS task scheduler is called when the system tick occurs, a blocking call or interrupt occurs (an interrupt that does invoke portYIELD_FROM_ISR()).
11. The call to the macro portYIELD_FROM_ISR() returns for some platforms, but not for others. You should always write the code to expect both.
12. The peripheral routine pcnt_counter_pause() merely sets a bit in a memory-mapped word. As such, it does not introduce any blocking or take long to execute. Bonus: note, however, that there is a very short critical section within the code to prevent conflict with the single bit.
13. When the queue is full, calling xQueueSendFromISR() returns pdFALSE (the call fails).
14. For platforms where portYIELD_FROM_ISR() does not return, the statements following it will not be executed.
15. The call to delay() is a blocking call. This Arduino function invokes the FreeRTOS routine vTaskDelay(). In an ISR, a blocking call is verboten.

Chapter 10 – Queue Sets

1. Yes but this is faulty design. After the resource handle has been returned, another resource operation on that handle must complete. The two steps represent a race condition.
2. Three queues were used so that if a given button has too many key bounce events and fills the queue, it will not prevent the other buttons from queuing an event.
3. The xQueueAddToSet() will fail if the queue to be added is not empty.
4. In the call to xQueueAddToSet(), the queue set handle is argument two.
5. When xQueueAddToSet() is properly called, it can return a failure when the added queue is not empty, the added binary semaphore has been given, or the added mutex has been locked.
6. The QueueSetMemberHandle_t data type can represent a handle to a queue, a binary semaphore, a counting semaphore or a mutex.
7. A queue set monitoring a queue of depth 3, a binary semaphore, and a counting semaphore with a count of 5 should have a minimum depth of 3 + 1 + 5 (9 entries).
8. Queue sets are *not* recommended for use with mutexes when task priority must always be respected. This is because the call xQueueSelectFromSet() does not boost a lower priority task when it owns the mutex being sought.

Chapter 11 – Task Events

1. The task event word is part of the Task Control Block (TCB).
2. Calling xTaskNotifyGive() increments the receiving task's event word.
3. The task event word is an unsigned 32-bit value, allowing 32 bits.
4. You might supply a non-zero ulBitsToClearOnEntry value when calling xTaskNotify-Wait() to clear bits that may have been handled as part of the last pass of an event loop. In this way, the call to xTaskNotifyWait() will block execution and wait for the next event rather than immediately return for a previously posted event that was just processed.
5. The function ulTaskNotifyTake() should be used in preference to xTaskNotifyWait() when you only require a simple wake-up call or a counting-like semaphore.
6. When eNoAction is used to notify a task, the task is notified without any change to its event word.
7. The value pdFALSE is returned when xTaskNotifyWait() times out.
8. The value returned from xTaskNotify() will return pdFAIL when it fails. The function can fail when eSetValueWithoutOverwrite fails to meet the requirements of the call.
9. When a task is blocked waiting for ulTaskNotifyTake() or xTaskNotifyWait(), no CPU time is consumed while the task is not able to be scheduled.

Chapter 12 – Event Groups

1. The EventBits_t data type can hold up to 24 event bits, even though the data type is 32-bits in size.
2. Using xEventGroupSetBits() and xEventGroupWaitBits() to synchronize doesn't work because the pair of operations are not atomic. One or more event bits may change between the call to xEventGroupSetBits() and the call to xEventGroupWaitBits(). The call to xEventGroupSync() performs the bit set and test atomically.
3. To act as a barrier, the event group bit must be initially zero (inactive). One or more tasks then wait upon that event bit to become a 1-bit, and then the bit is *not* cleared. As long as the event bit remains a 1-bit, it signals the release of the barrier.
4. A non-barrier event bit is cleared after it has been sensed active. This causes future xEventGroupWaitBits() calls to block until the event is re-activated.
5. Zero to an unlimited number of tasks may block on an event group, limited only by memory.
6. The handle becomes invalid, but the handle's value remains unchanged. Reusing a deleted handle is dangerous because it points to memory that has been freed. The application designer should reset the handle to nullptr (NULL) after the resource has been deleted, for safety.
7. The return value from xEventGroupWaitBits() should be checked to see that all of the required event bits were set. If they were not, the return from the function was due to a timeout.
8. When xWaitForAllBits is set to pdFALSE, any event bit is accepted (logical OR of the required bits).

Chapter 13 – Advanced

1. A critical section permits a short but critical sequence of code to execute uninterrupted.
2. Even when software is perfect, a watchdog timer may save the application when a hardware issue like a peripheral has got hung up. A reset and boot may clear the problem and restore the application's function.
3. Having the watchdog enabled for the Idle task permits the watchdog to catch scheduling and priority issues. If the Idle task never gets CPU time, then certain functions like the timers cannot execute.
4. Setting global boolean loopTaskWDTEnabled to true only enables the call to esp_task_wdt_reset() within the loopTask. To actually enable the watchdog, you need to call upon esp_task_wdt_add() in addition to setting loopTaskWDTEnabled to true.
5. Task local storage avoids making its values globally accessible. It can also be useful when multiple tasks run with different values.
6. A task that calls vTaskSuspendAll() does continue to execute but be careful. If it invokes other routines that depend upon the scheduler, there can be a problem.
7. The function xTaskGetTickCount() returns the current time in system ticks. On the ESP32, this happens to be 1-millisecond units.
8. The function xTaskGetTickCount() value can overflow and wrap around. When that occurs, the later time will be lower in value than the earlier time.

Chapter 14 – Gatekeeper Tasks

1. If the user task accessed the gatekeeper before it was ready, it would have not had a message queue to send its request on. The event group prevents the caller from trying to send a message before the gatekeeper task has finished its initialization.
2. The gatekeeper API offers one or more of the following advantages:

    1. The API provided a mapping between a virtual GPIO number (XGPIO) and the selected I2C chip and pin number.
    2. It serializes the requests with the gatekeeper's message queue.
    3. It prevents clashes in simultaneous I2C bus transactions from independent tasks.
    4. A central place to manage the I/O port state.
    5. The requests can be checked for validity.
    6. Unit testing saves time and improves confidence in the services provided.

3. The gatekeeper approach centralizes the accessing code into one place. This permits unit testing to be performed on the gatekeeper alone, eliminating many application variables. Once the gatekeeper code is verified, the application development can focus on application issues.
4. The gatekeeper approach allows each request to be rigorously checked for validity.
5. The gatekeeper task approach requires a separate task to be allocated and a task priority assigned. Each task added adds to the memory consumption for stack use.

Appendix B – Parts

- ESP32 Lolin with OLED (dev board)
- (3) 220 ohm, 1/8 watt, 10% tolerance
- (3) 5 mm or 3 mm LEDs (red, green, yellow)
- (1) 10 Kohm linear potentiometer
- Assortment of Dupont wires: 10 mm or longer
- (3) individual push buttons or PCB with 4 push buttons with common ground.
- 3.3V to 5V level converter (minimum of 2 lines) for Chapter 5
- Ultrasonic Distance Sensor – HC-SR04 https://www.elektor.com/ultrasonic-distance-sensor-hy-srf05-160044-71 or equivalent.
- Adafruit Si7021 Temperature & Humidity Sensor Breakout Board or equivalent
- SparkFun Triple Axis Magnetometer Breakout – HMC5883L or equivalent
- (minimum 2) PCF8574 DIP (or equivalent module). Buy extras, in case they develop ESD damage from improper handling or nosey cats.
- Optional: TTGO ESP32 T-Display
- Optional: M5Stack

Index

FreeRTOS for ESP32-Arduino

Practical Multitasking Fundamentals

Warren Gay

Warren Gay is a datablocks.net senior software developer, writing Linux internet servers in C++. He got involved with electronics at an early age, and since then he has built microcomputers and has worked with MC68HC705, AVR, STM32, ESP32 and ARM computers, just to name a few.

Programming embedded systems is difficult because of resource constraints and limited debugging facilities. Why develop your own Real-Time Operating System (RTOS) as well as your application when the proven FreeRTOS software is freely available? Why not start with a validated foundation?

Every software developer knows that you must divide a difficult problem into smaller ones to conquer it. Using separate preemptive tasks and FreeRTOS communication mechanisms, a clean separation of functions is achieved within the entire application. This results in safe and maintainable designs.

Practicing engineers and students alike can use this book and the ESP32 Arduino environment to wade into FreeRTOS concepts at a comfortable pace. The well-organized text enables you to master each concept before starting the next chapter. Practical breadboard experiments and schematics are included to bring the lessons home. Experience is the best teacher.

Each chapter includes exercises to test your knowledge. The coverage of the FreeRTOS Application Programming Interface (API) is complete for the ESP32 Arduino environment. You can apply what you learn to other FreeRTOS environments, including Espressif's ESP-IDF. The source code is available from github.com. All of these resources put you in the driver's seat when it is time to develop your next uber-cool ESP32 project.

What you will learn:

- How preemptive scheduling works within FreeRTOS
- The Arduino startup "loopTask"
- Message queues
- FreeRTOS timers and the IDLE task
- The semaphore, mutex, and their differences
- The mailbox and its application
- Real-time task priorities and its effect
- Interrupt interaction and use with FreeRTOS
- Queue sets
- Notifying tasks with events
- Event groups
- Critical sections
- Task local storage
- The gatekeeper task

ISBN 978-1-907920-93-6

9 781907 920936

Elektor International Media BV
www.elektor.com