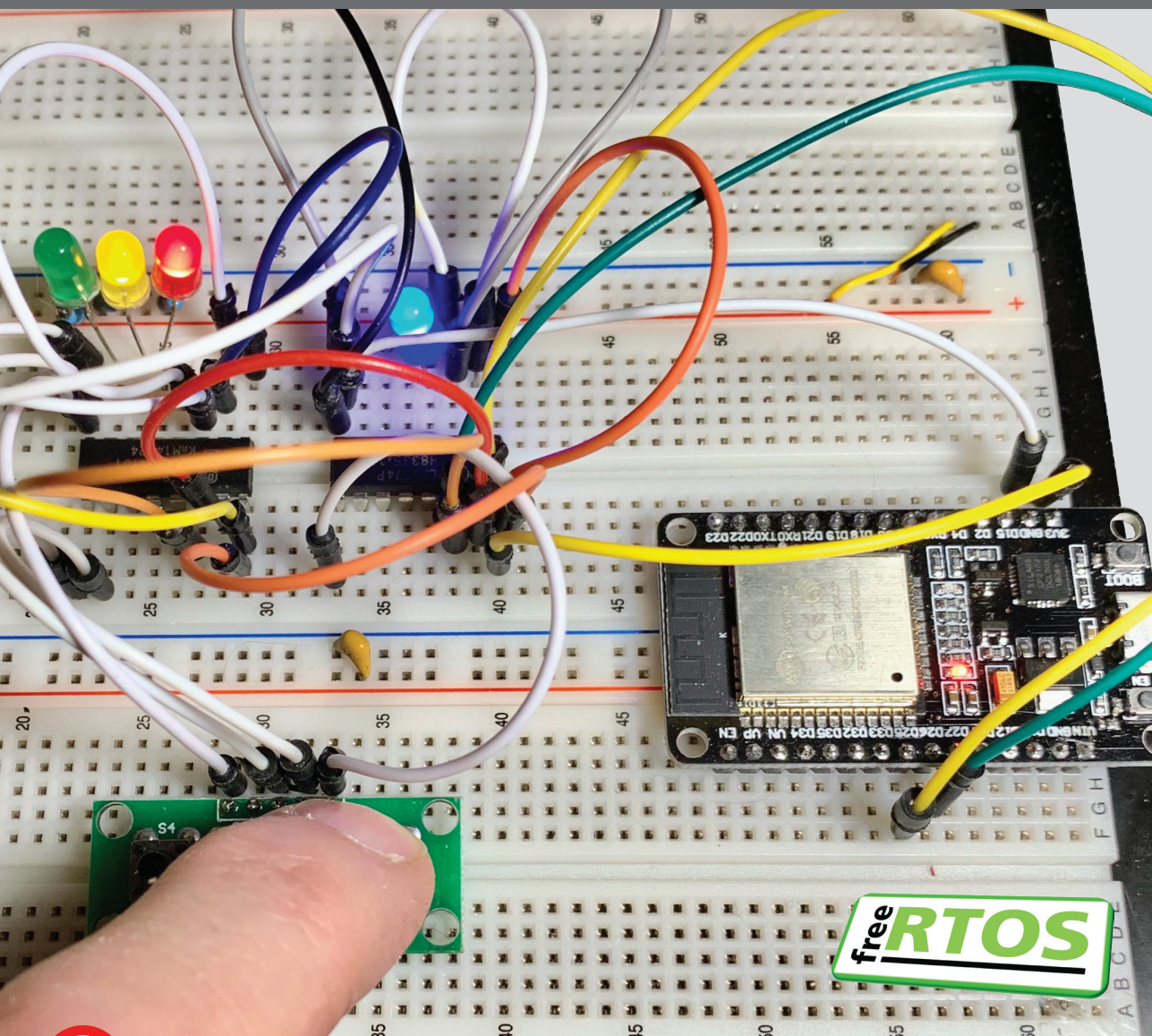
FreeRTOS for ESP32-Arduino

Practical Multitasking Fundamentals



elektor

Warren Gay

FreeRTOS for ESP32-Arduino

Practical Multitasking Fundamentals

Warren Gay

elektor

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

● Declaration

The author and publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, or hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other cause..

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

● **ISBN 978-1-907920-93-6**

Prepress Production: D-Vision, Julian van den Berg
First published in the United Kingdom 2020

Chapter 1 • Introduction 17

The Need for RTOS 17

FreeRTOS Engineering 18

Hardware 19

Dev Boards 19

ESP8266 20

FreeRTOS Conventions 20

Variable Names 21

Function Names 21

Macro Names 21

Header Files 21

Arduino Setup 22

ESP32 Arduino 22

ESP Related Arduino Resources 24

C and C++ 24

FreeRTOS and C++ 25

Arduino FreeRTOS Config 26

ESP32 Notes 27

Arduino GPIO References 27

Input Only 27

Reserved GPIOs 27

GPIO Voltage and Drive 28

Programs 28

Graphics/Drivers Used 28

TTGO ESP32 T-Display 28

M5Stack 29

Assumptions about the Reader 29

Summary 29

Web Resources 29

Chapter 2 • Tasks 30

Preemptive Scheduling 30

Arduino Startup 31

Main Task 32
Task Demonstration 33
Program Design 35
Stack Size 37
Memory Management in FreeRTOS 38
Static Tasks 39
Task Delete 41
Task Suspend/Resume 43
Task Time Slice 44
Yielding CPU 47
Assert Macro 49
Summary 51
Exercises 51
Web Resources 51
Chapter 3 • Queues 52
Queue Characteristics 52
Arrival Pattern 52
Capacity 53
Service Discipline 53
Sources and Destinations 53
Basic Queue API 54
Creating Static Queues 54
Queuing an Item 55
Receiving from a Queue 56
Dynamic Queue Creation 57
Queue Delete 57
Queue Reset 57
Task Scheduling 58
Blocked while Adding 58
Blocked while Receiving 58
Demonstration 59
Program Setup 60

Debounce Task 63
LED Task 63
Press Demonstration 63
Safety Improvement 68
The Temptation to Optimize 72
Informational API 72
Peeking at the Queue 72
Variable Length Items 73
Interrupt Processing 73
Summary 74
Exercises 74
Chapter 4 • Timers 75
Timer Categories 75
Software Timers 76
The Timer Callback 76
Timer Limitations 76
Timer ID Value 77
Abusing Timer ID 78
Timer Types 79
Timer States 79
Create Static Timer 80
Create Dynamic Timer 81
Activating the Timer 81
Demonstration 82
AlertLED Constructor 84
AlertLED Instance 84
AlertLED::alert() Method 85
Stopping the Alert 86
setup() and loop() 86
Demo Notes 89
Priority 1 89
The Class Advantage 89

Task Timer API 90
xTaskGetTickCount() 90
xTaskDelayUntil() 91
Demonstration Observation 94
Summary 96
Exercises 96
Web Resources 96
Chapter 5 • Semaphores 97
Semaphore Types 97
Binary Semaphores 97
Counting Semaphores 98
Binary Semaphore Demonstration 98
Program Operation 105
Locks 106
Deadlocks 106
Dining Philosophers 107
Dining Philosophers Demo 107
Deadlock Prevention 109
Lockups 109
Insidious Deadlocks 114
Summary 115
Exercises 115
Web Resources 115
Chapter 6 • Mailboxes 116
The Problem 116
The Mailbox 117
Creating a Mailbox 118
Reading the Mailbox 118
Mailbox Demonstration 118
Program Dissection 121
Summary 129
Exercises 129

Web Links 129
Chapter 7 • Task Priorities 130
vTaskStartScheduler() 130
What does vTaskStartScheduler() do? 130
Configured Scheduling Algorithm 130
Task Pre-emption 131
Time Slicing 131
ESP32 Task Priorities 131
Task States 131
I/O and Sharing the CPU 132
Preventing Immediate Task Start 133
Simple Demonstration 133
Blocking 135
Creating a Ready-to-Go Task 137
ESP32 Dual Core Wrinkle 139
Priority Demonstration 140
Experiment 1 140
Experiment 2 142
Experiment 3 142
Experiment 4 143
Priority Configuration 149
Scheduler Review 149
Summary 150
Exercises 150
Web Resources 150
Chapter 8 • Mutexes 151
Exclusion Principle 151
What's the Problem? 151
The Mutex Solution 152
Priority Inversion 153
Creating a Mutex 154
Give and Take 154

Deleting a Mutex 155
Demonstration 155
PCF8574 Chip 155
LED Drive 156
Code Break Down 157
Troubleshooting 158
Blink Loop 158
Running the Demonstration 159
Recursive Mutexes 163
Recursive Mutex API 163
Deadlock Avoidance and Prevention 164
Recursive Mutex Usage 164
Summary 165
Exercises 165
Web Resources 165
Chapter 9 • Interrupts 166
Characteristics of an ISR 166
The Asynchronous ISR 166
The ISR Stack 167
Non-Reentrant Routine Calls 167
ISR Priorities 167
Short ISR Routines 168
ISR is not a Task 168
Special ISR Code 168
ESP32 Arduino GPIO Interrupts 169
Frequency Counter Project 169
Challenges 169
Approach 170
Case 1 – 300,000 Hz 171
Case 2 – 500 Hz 171
Project Code 172
Range Finding 174

ISR Routine 175
xQueueSendFromISR() 176
portYIELD_FROM_ISR() 177
Running the Demo 177
Troubleshooting the Wemos Lolin ESP32 179
Pulse Counter Notes 179
Setup for Interrupts 180
TTGO ESP32 T-Display 186
Troubleshooting the TTGO 187
M5Stack. 188
Troubleshooting M5Stack 189
Summary 190
Exercises 190
Web Resources 190
Chapter 10 • Queue Sets 191
The Problem 191
The Queue Set 192
Queue Set Configuration 192
Queue Set Select 192
Queue Set Traps To Avoid 193
xQueueAddToSet Trap 1. 193
xQueueAddToSet Trap 2. 193
Demonstration 194
Program Breakdown 195
setup() 196
ISR Routines 197
Event Monitoring Task 198
Mutexes 199
Summary 202
Exercises 202
Chapter 11 • Task Events 203
Task Notification 203

Restrictions 203
Waiting 204
ulTaskNotifyTake() 204
Binary Notification 204
Counting Notification 205
Give Notify 205
Demonstration 1 205
Demonstration 2 208
Demonstration 3 210
Going Beyond Simple Notify 212
Argument 3 – ulBitsToClearOnExit 213
Argument 2 – ulBitsToClearOnEntry 213
Smart Notify 213
Argument eAction 213
Demonstration 4 214
Demonstration 5 217
Summary 222
Exercises 222
Web Resources 222
Chapter 12 • Event Groups 223
EventBits_t Type 223
Creating an Event Group Object 223
Notifying an Event Group 224
Waiting for Event Groups 224
Demonstration 1 225
Demo Conclusion 229
Synchronization 236
Demonstration 2 236
Auxiliary Functions 241
vEventGroupDelete() 241
xEventGroupClearBits() 241
xEventGroupGetBits() 241

xEventGroupSetBitsFromISR() 242
Summary 242
Exercises 242
Web Resources 242
Chapter 13 • Advanced Topics 243
Watchdog Timers 243
Watchdog Timer for the loopTask 243
Enabling Task Watchdog 245
Watchdog For Multiple Tasks. 247
Non-Arduino Watchdog Use 251
The Idle Task 252
Critical Sections 252
ESP32 Critical Sections 253
Critical Sections for ISRs 255
Interrupts 255
Task Local Storage 255
uxTaskGetNumberOfTasks() 259
xTaskGetSchedulerState() 259
eTaskGetState() 259
xTaskGetTickCount() 259
vTaskSuspendAll(). 259
ESP32 Arduino Limitations 260
Summary 260
Exercises 260
Web Resources 260
Chapter 14 • Gatekeeper Tasks 261
Gatekeepers 261
Demonstration 262
Extension GPIO Designations 262
Gatekeeper API. 263
Demonstration XGPIO 263
Operation 265

Gatekeeper Code 265
Gatekeeper Initialization 267
Gatekeeper API Functions 267
Gatekeeper Task 270
Input from PCF8574P 270
Output to the PCF8574P. 271
PCF8574P State Management 271
Troubleshooting 272
Summary 280
Exercises 281
Chapter 15 • Tips and Hints 282
Forums: Invest Some Effort 282
Start Small. 282
The Government Contract Approach 283
The Basic Shell 283
The Stub Approach 283
Block Diagrams. 284
Faults 285
Know Your Storage Lifetimes 285
Avoid External Names 286
Leverage Scope 286
Rest the Brain. 287
Note Books. 287
Asking for Help 287
Divide and Conquer. 288
Programming for Answers 288
Leverage the find Command. 289
Infinitely Malleable 291
Make Friends with Bits 292
Efficiency 292
Source Code Beauty 292
Fritzing vs Schematics 293

Pay now or Pay Later 293
Indispensable Programmers 293
Final Curtain 294
Appendix A 295
Appendix B – Parts 301
Index 302

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 (,).

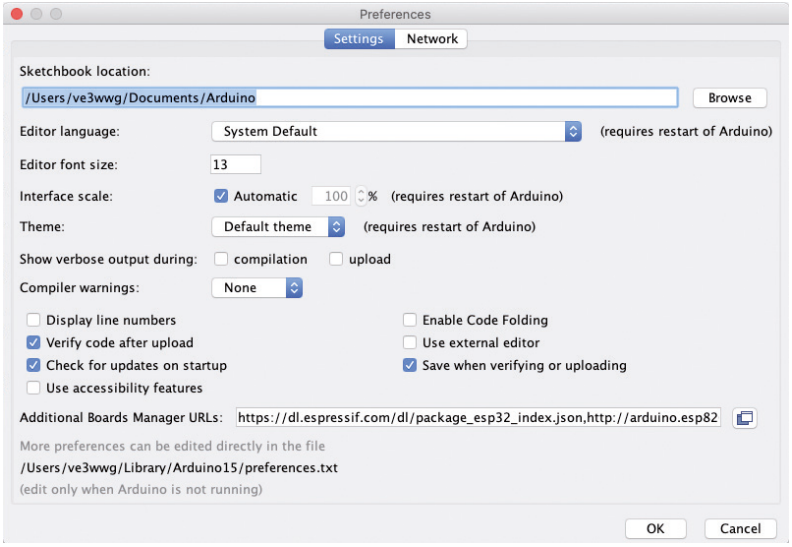


Figure 1-1. File->Preference dialog.

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

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

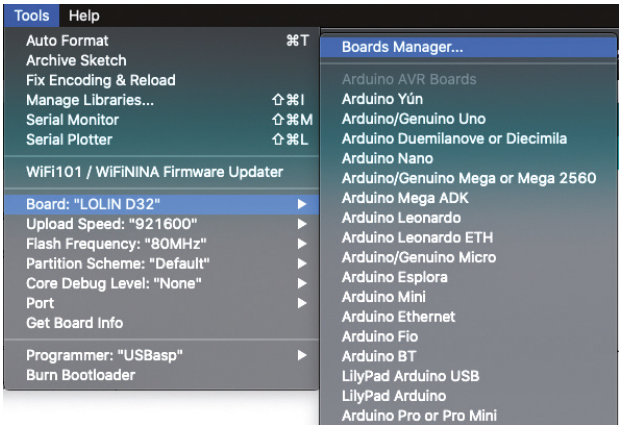


Figure 1-2. File->Board->Boards Manager menu selection.

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.

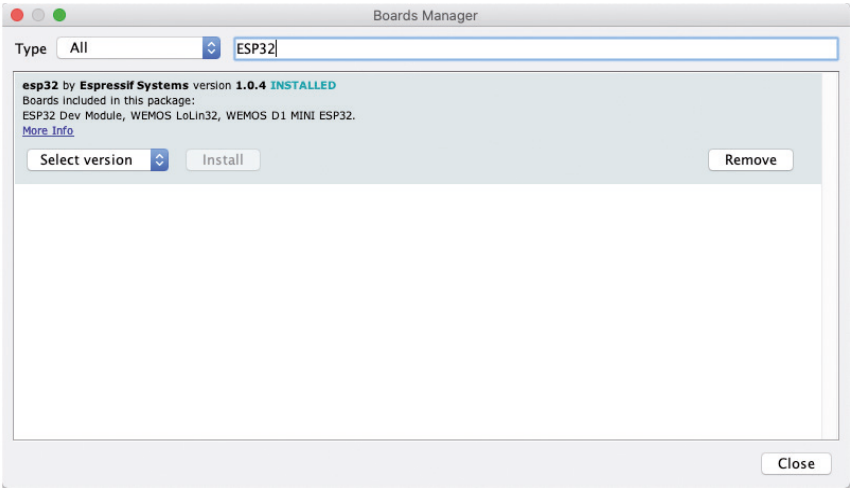


Figure 1-3. Boards Manager after the ESP addition is installed.

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 dec-

larations. 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/

Index

Symbolen
{} 290
+3.3 volts 99
~ 290
$HOME 290
2>/dev/nul 290
3.3-volt GPIO ports 28
3.3-volt regulator 19
5-volt device 99
5 volts tolerant 28
5-volt tolerant GPIOs 99
"%s" format item 285
"\;" token 290
/usr/bin/nc 228
__volatile__ keyword 141

A
abusing the pointer 78
active high 60
active low 264
active low configuration 156
Adafruit Si7021 Library 119
ADC 178, 186, 189
ADC input value 178
Additional Boards Manager URLs 22
address pins 157
AlertLED 82
allocated on the stack 286
analog voltage 177
anti-static (ESD) wrist band 272
API 17, 203
APP_CPU 139
Application CPU 139
application tasks 31
app_main() 32
Arduino 17, 29, 288
Arduino compiler options 285
Arduino environment 31
Arduino IDE 22
Arduino Serial Monitor 19
Arduino startup code 32
array 35
array extents 36
assert.h 49
assertion 118, 264, 272
assertion error 50
assertion fault 264
assert() macro 55
associated "user data" 77
assuming 292
asynchronous 77
asynchronous event 76
asynchronous routine 166
atomic 53, 116, 117, 122, 129, 236
atomically 18
atomicity 117
attachInterrupt() 169
automatic range finding 170
auto-reload 80
auto-reload timer 79
avoid locking 164
avoid overflowing the size of the 32-bit register 173

B
bad practice 164
balking 53
Balking 52
Bankers Algorithm 164
barrier 97, 226, 229
BaseType_t 20
basic shell 283
baud 28
bicolour LED 288
binary semaphore 98, 105, 106, 119, 122, 133, 152, 153, 154, 1 57, 165, 203, 204
Binary semaphore 155
binary semaphore as a lock 108
binary semaphores 151, 192
binary semaphore take operation 204
bit fields 268
BIT(x) 292
block 97, 132, 149, 192, 203
block diagram 284
blocked 53, 55, 58, 76, 149, 152, 204, 206
Blocked 132
Blocked state 132

blocking calls 150
blocks 135, 143, 150, 152, 205, 228
blocks forever 58
Bluetooth 31
Board Manager 23
boost the lower priority task 152
boost the priority 153
Bouncing metal contacts 63
broadcast address 229
bus transactions 155
busy-wait loop 132

C
C++ language 24
C++ objects 73
C++ reference 36
C++ startup 32
C++ std::map 255
C++ templates 291
cache 117
callback 77
capacity of a queue 53
C assert macro 49
Catalina 22
CH340 19
change semaphore 122, 123
circular dependencies 164
C language structure initialization 24
ClearOnExit 224
client tasks 266
code 288
code aborts 37
code coverage 37
code maintenance 197
code smell 165
compass 119
compass readings 123
compiler 193
computes the frequency 171
concurrent 54
concurrent execution 18
concurrent processing 30
CONFIG_FREERTOS_ISR_STACKSIZE 167
configMAX_PRIORITIES 149
consume CPU time 141
consumers 52
cooperative multitasking 131
counter peripheral 170
counting semaphore 98, 109, 210, 212
Counting Semaphore 155
counting semaphores 97, 107, 192
cowboy programming style 286
CP2102 19
CP2104 19
CPU 0 31, 36, 139
CPU 1 31, 36
CPU cycles 141
CPU number 36
CPU starved 150
CPU time 37
CPU time fairness 131
create a timer 80
create tasks before starting the scheduler 130
creation of the mailbox 118
critical.ino 253
critical section 18
critical sections 243
current strengths 28
custom GPIO values 158

D
daemon task 77
damon task 89
data item lifetime 56
Data RAM 38
deadlock 106, 107, 110, 114, 164
Deadlock 109
deadlock avoidance 164
deadlock detection 164
deadlock prevention 114
deadlock prevention technique 107
deadlocks 18
deadly embrace 106
debounce 63
debounced 60
debouncing 64
debug 89, 283
debugger 283
debugging 60
debugging serial link 283
debugging sessions 283

delay 86, 136, 247
delay() 35, 41, 60, 75, 76, 90, 158, 283
delete (terminate) a task 41
deleting self 41
dev board 19, 28, 159, 194, 265, 272, 288
device index 270
devx 270
Digital Multi Meter 100
Dining Philosopher's problem 107
D/IRAM 39
directly notify 203
disabled interrupts 253
disable interrupts 255
disabling interrupts 253
disconnect 226
divide and conquer 288
dividing up CPU 144
DMM 100
dormant 80, 81
Dormant 79
dormant state 81
driver 179
dual-core ESP32 30
dynamically allocated queue 57
dynamically allocated timers 81

E

eAction 213
eBlocked 259
Echo signal 100
eDeleted 259
efficiency 56, 292
efficient 291
eIncrement 213
elapsed time 176
embedded devices 283
empty 97
empty mailbox capability 120
empty queue 57, 59
eNoAction 213
errQUEUE_EMPTY 56
errQUEUE_FULL 55, 63
eRunning 259
ESD (Electrostatic Discharge) 28
ESD preventative measures 273
ESD protection 272
ESD protection diodes 170
eSetBits 213
eSetValueWithoutOverwrite 213
eSetValueWithOverwrite 213
ESP32 64-bit high-resolution timer 170
ESP32CAM 19
ESP32 Lolin with OLED 19
ESP32 Oled Driver 28
ESP32-S 30
ESP32-S2 30
ESP32 support 22
ESP32 system mutex 253
ESP32 Wemos Lolin device 159
ESP8266 20
ESP hardware timer API 75
ESP-IDF 20, 39, 252, 260, 288
Espressif 29
Espressif product offerings 20
esp_task_wdt_add() 246
esp_task_wdt.h 245
esp_task_wdt_init() 246, 247
esp_task_wdt_reset() 244, 246, 247
ESP Technical Reference Manual 180
eSuspended 259
eTaskGetState 259
EventBits_t 224, 225
EventBits_t Type 223
event flags 223
eventgr.ino 225
event group 98, 203, 223, 225, 229, 236, 241, 267
event groups 203, 223
event notification 203, 229
events 192, 203, 285
event word 212
evsync.ino 236
excessive stack nesting 168
exclusion principle 151
exec 290
execution 132
execution priority 130
execution state of a task 132
extender chip 158
extern 286

external pull-up resistor 194
external symbol 286
extern "C" 25, 32

F
fatal error 43
fault 285
FIFO 56
FIFO queue 53
file globbing 291
find 290
find command 289
find command's name option 291
five heap implementations 38
fixed time slice 131
Floating point 173
floating potential 169
flow control 53
Forums 282
fragmented 57
free() 39, 73
FreeBSD 289
FreeRTOS 17
FreeRTOS configuration 26
FreeRTOS manual 20
FreeRTOS Reference manual 260
FreeRTOS software timer support 75
FreeRTOS task priority 168
freqctr-m5.ino 188
freqctr-ttgo.ino 187, 188
frequency counter 169, 170, 178
frequency measurement 169
Fritzing diagrams 293
FromISR 176, 190
FromISR() 149
FromISR suffix 215
FT232RL 19
full 97
full queue 58
full slice 46, 47
full stack allocation 167
full-time slice 131

G
gatekeeper API and task 266
gatekeeper.ino 262, 265
gatekeepers 261
gatekeeper task 261, 262
generated frequency 178
generic code 291
give 97, 154
given 97, 154, 157
given binary semaphore 194
gives 98
giving 151
Giving 98
global value that is null 118
global variables 116
government contract approach 283
GPIO 170
GPIO 0 19
GPIO expanders 158
GPIO extender 155
GPIO extender chips 157, 262
GPIO interrupt processing 195
graphic driver 28
graphics driver support 186
grep 290
grep command 290
grep regular expression 290
group notifications 223

H
hall effect sensor 247
hallRead() 108
hammered with interrupts 176
handle 54, 77, 198, 202, 224
handles 286
hangs 28
hardcore priority 130
hardware timer 30
HC-SR04 100, 106
HC-SR04 module 99
header files 21
heap 57
heap bytes available 38, 42
heap_caps_free 39
heap_caps_malloc() 39
high-resolution hardware timers 76
high water mark 38
hints 282
HMC5883L 118, 119

HomeBrew 228
H option 290
http_server() 226
humidity 122

I

I2C 119, 157, 158, 165, 287
I2C address 156, 157, 262, 270, 271
I2C bus 119, 122, 151, 152, 153, 155, 156, 158, 163, 165, 261, 262
I2C devices 151, 261
I2C expander chip 157
I2C problem 272
I2C temperature sensor 165
I2C transaction 158, 263
I2C transmission 272
IDE issues 24
IDLE1 45
idle task 130, 149
Idle task 252
IDLE task 43
Idle task in Arduino 252
incoming signal frequency 170
initArduino() 32
initialize the semaphore 106
initial state of a created task 132
inlined function 197
inline functions 291
inline keyword 197
input only 169
input protection diodes 170
INPUT_PULLUP 169
input signal 170
Input signal conditioning 169
Instruction RAM 39
Instruction Random Access Memory 168
interface 262
internal linked list of memory blocks 167
interrupt 18, 149, 166, 169, 177, 193
Interrupt 180
interrupted code 167
interrupted task 167
interrupt flag 168, 176
interrupt priorities 1 to 3 149
interrupt priority 168
interrupts 143, 168, 169, 190, 217, 253
Interrupts 166
Interrupt Service Routine 166
interrupts for each CPU core 167
interrupt status word 176
INT pin 156
I/O peripheral event 132
ipc1 45
IP number 227
IRAM 168
IRAM_ATTR 168, 190
IRAM memory 169
ISR 73, 166, 170, 173, 176, 177, 179, 191, 193, 194, 203
ISR code 168
ISR for GPIO interrupts 169
ISR handler 196
ISR (Interrupt Service Routine) 53
ISR is suspended 168
ISR routine 167, 175, 177
ISR routines 168, 190, 197, 198, 217, 284
ISR specific macros 255
ISR stack 166
ISR stack convention 167
ISR tracing 288

L

lambda functions 174
LED 288
LED indicators 218
LEDs 33, 282
level converter 100
library nesting issue 164
lifetime of the object 73
limitations for GPIO 27
limit control devices 98
limit the scope 286
Linux 228, 289
listen for UDP packets 229
local storage 258
lock 44, 106, 152, 153, 163, 165
lock a mutex 154
locking semaphore 122
lock multiple locks 164
lock nesting problem 165
locks 153
Lolin 32 ESP 140

Lolin ESP32 99
Lolin module's OLED display 119
loop() 31, 32, 41, 84, 86, 131, 150, 205, 208, 247, 283, 284, 286
loopTask 31, 32, 37, 41, 45, 49, 131, 133, 137, 138, 141, 236, 243, 247, 249, 251, 284
loopTask() 205, 236, 244
loopTaskWDTEnabled 244, 245
lowest execution priority 31

M

M5Stack 29, 169, 188, 189
MacOS 19, 22, 228, 289
macro 197
Macro names 21
macro procedures 291
magic smoke 287
mailbox 116, 117, 118, 119, 122, 123
mailboxes 123
mailbox.ino 120
main Arduino task 31
main task 41
make menuconfig 39
malloc() 167
malloc() and free() 38
MALLOC_CAP_* macro values 39
maximum queue depth 55
MCU 30
member handle 202
memory corruption 56, 291
memory leak 73
memory leaks 291
memory-mapped register 180
memset() 25
Microcontroller Unit 30
micros() 76, 170
microsecond timer 176
microsecond timer value 176
millis() 75, 90
monitoring task 198
monopolizing the CPU 30
multi-core support 37
multiple CPU problem 117
multiple locks 164
multitasking friendly 133
Murphy's law 114
mutex 18, 106, 108, 151, 152, 153, 154, 155, 157, 158, 163, 165, 194, 198, 202, 203, 253, 286
mutex boosts 153
mutexes 97, 192, 199, 261, 285
mutual exclusion 151

N

namespace 196
naming conventions 20
nc 228
nested calls 167
nested ISR routine calls 167
nested lock count 163
nested locking 165
netcat 229
netcat command 228
netcat.exe 229
newbie 282, 289
newlib 285
newlib library 28, 38, 167
non-empty mailbox 118
non-reentrant 166, 167
no priority boost 199
notify call 203
Notifying an event group 224
not ready state 58
nRF24L01 library 290
NULL 24, 35, 37, 55, 57, 176, 192, 212, 256
nullptr 24, 35, 37, 40, 42, 55, 57, 77, 81, 176, 192, 212, 256, 258, 285

O

object lifetimes 285
observable strangeness 288
OLED 19, 28, 99, 104, 119, 140, 159, 169, 173, 177, 178
OLED display 178
OLED display driver 179
OLED I2C address 140
one-shot timer 79
optimizations 116
order of operations 292
Output GPIOs 28

overly locked mutex 163
owner 152, 153
owns 153

P
P0 to P7 270
packaged software 163
panic reboot 246
parity 28
PCF8574 155, 156, 157, 158, 159
PCF8574A 155, 156, 157, 264
PCF8574P 262, 264, 267, 270, 271, 272, 273
PCF8574P DIP pinout 262
PCF8574P GPIO extender chips 261
PCF8574P GPIO pins 262
PCF8575P 262
pcnt_counter_pause() 179
pdFAIL 82, 97, 154, 193, 213
pdFALSE 81, 205, 210, 212, 224, 260
PDIP 155
pdMS_TO_TICKS() 80
pdMS_TO_TICKS(ms) 55
pdPASS 55, 56, 82, 117, 154, 213
pdTRUE 81, 204, 205, 206, 212, 224, 228, 260
peek 123
ping 105
pinMode() 169
Pinout of the PCF8574/PCF8574A chip 156
plain mutex 164
Plastic Dual Inline Package 155
POD (Plain Old Data) 26
polling 17
portability 20, 27
portDISABLE_INTERRUPTS 255
portENABLE_INTERRUPTS 255
portENTER_CRITICAL 253
portENTER_CRITICAL_ISR 255
portEXIT_CRITICAL 253
portEXIT_CRITICAL_ISR 255
port index 270
portMAX_DELAY 55, 58, 59, 63, 67, 82, 105, 108, 224
portMUX_TYPE 253, 255
portx 270
portYIELD_FROM_ISR() 177
POSIX system 289
potentiometer 178
precedence of C operators 292
preempted 30
pre-emption 131
preemptive context changes 45
preemptive scheduling 18, 131
pre-empts 153
print() 86
printf() 28, 38, 60, 77, 89, 285
priorities 151
priority 18, 30
priority-based interrupts 167
priority-based queue 53
priority inversion 152, 153, 199
priority levels 131
priority numbers range 149
PRO_CPU 139
producers 52
program fault 285
programming 19
programs 28
protected resource 151
protection diode 28
protocol 151
Protocol CPU 37, 139
provision for interrupt processing 73
pull-up 27, 271
pull-up resistances 194
pull-up resistor 169
pullup resistor 60
pull-up resistor/regulator circuit 155
pull-up resistors 27
pulNotificationValue 213
pulse 45
pulse counter 170, 176
pulse counter peripheral 169, 180
pulseInLong() 106
Pulse Width Modulation 170
push button and switch inputs 27
push button input GPIO 60
pvTaskGetThreadLocalStoragePointer 256
pvTimerGetTimerID() 78
PWM 170, 177, 178, 179, 187, 189, 190
PWM Arduino API 170

PWM frequency 178
pxHigherPriorityTaskWoken 215

Q
qset.ino 195
qualities of an ISR 166
quasi-bidirectional device 271
query the current event group bit values 241
queue 52, 117, 118, 132, 141, 152, 153, 191, 192, 194, 196, 198, 202, 203, 267, 286
queue depth 53, 57, 60
queue depths 192
queue handle 56
queue is full 58
queue object 54
queue overwrite 129
queue peek 129
queues 191, 192, 199, 261, 285
queue set 191, 194, 196, 198
QueueSetHandle_t 192
queue sets 199
Queue sets 202
Queue Set Traps 193
quirk of the PCF8574 chip 156

R
race condition 198
race conditions 291
rand() 167
random seed value 108
rand_r() 167
Range finding 169
Ready 132
ready state 58
Ready state 131, 132, 139, 140, 149
ready task 131, 149
Real-time priority 150
Rebooting 249
recursively 167
recursive mutex 163, 164, 165
recursive Mutex 155
reentrant 167
reentrant routines 167
reference 196
register 116, 117
register optimizations 117
registers 30
register save and call convention 168
relative humidity 120
releasing the mutex 153
reneging 53
Reneging 52
reset 150
restart 150
resumption 44
RF24.h 290, 291
RISING 169
rising edge 169
rising pulse 169
risks 165
round-robin 48, 49, 131, 133, 142, 149, 150
Round-Robin 139
round-robin scheduling 49, 139
round-robin unfairness 143
routine in one place 197
RTOS daemon 252
RTOS daemon task 77, 241
running 80
Running 79, 132
running out of stack space 285
Running state 132

S
safety-critical applications 268
scheduler 30, 139, 149, 177, 197, 253
scheduler component of FreeRTOS 139
scheduling 131
scheduling can be unbalanced 143
schematic 119, 156, 177, 194, 205, 264
Schematic diagrams 293
SCL 156, 157, 272
scope capture 46
scope rules 286
scoping rules 291
SDA 156, 157, 272
SDK 39
self 42
semaphore 97, 119, 152, 198, 202, 203
semaphores 191, 261, 285

send a pointer through a queue 285
sending data items by pointer 73
sensor modules 119
sequencing of the locks 114
serial interface 28
serial monitor 37, 227
Serial monitor 60
Serial Monitor 28, 99, 104, 106, 107, 119, 123, 133, 135, 158, 159, 205, 208, 218, 225, 246, 249, 264, 272, 284, 288
serial monitor output 35
Serial Monitor output 134, 135
setup() 31, 32, 35, 36, 60, 67, 84, 86, 97, 105, 108, 121, 131, 133, 136, 141, 150, 157, 193, 196, 198, 205, 246, 267, 283, 284, 286
Setup25_TTGO_T_Display.h 187
share one ISR routine 197
share the CPU 48
Si7021 118, 119
Si7021 sensor 119
signal generator 169
signal level translations 99
silencing of the speaker on the M5Stack 189
simultaneous access 151
single or multiple queuing sources 53
sink current 156
sinked 264
sketch 28
slave devices 151
Smart Notify 213
SMP 139
snprintf() 77
Soc 17
sole owner 151
source current 156
sourced 264
special calling requirements 166
special case of the queue 129
spin lock 253
spinning 47
SPI RAM 39
SRAM 17, 19, 60
SSD1302 140
SSD1306 28, 140, 177
SSD1306 OLED 99
stack 17
stack bytes 35
stack for ISR routines 167
stack high water mark 38
stack size 37, 40
Stack size 36
stack size limitations 76
stack space 77
StackType_t 40
stack usage 38
starts the FreeRTOS scheduler 130
state machine 17
states of a FreeRTOS task 132
static 286
static electricity 28
"static" keyword 40
static keyword 286
static queue 55
static queues 54
StaticTimer_t 80
std::string 73
stop bit 28
strdup() 73
struct 35, 196
stubbing out your application 285
stub functions 283
suffix "FromISR" 73
support tasks 31
suspend 39
suspend and resume tasks 43
suspended 149, 150
Suspended 132
Suspended state 132
suspending 137
suspending a task 44
suspension 44
symmetric multiprocessing 139
synchronization 97, 254
synchronize 97
synchronizing 98
synchronous event 52
system tick 177
system tick interrupt 131, 142, 149, 168, 253

system timer ticks 131

T

take 97, 98, 154, 158
taken 97, 151, 154
taking 98
task 30, 117, 131, 166, 203
task context change 139
Task Control Block 39
Task Control Block (TCB) 38
task creation 31
task deletes another 43
task deletes self 43
taskENTER_CRITICAL() 253
task event notification 203
task event word 204
task execution 203
taskEXIT_CRITICAL() 253
task handle 37, 40, 42, 44, 258
task handles 35
tasklocal.ino 256
task local storage 243
tasknfy1.ino 207
tasknfy3.ino 212
tasknfy4.ino 214
tasknfy5.ino 219, 222
task notifications 223
task notification word 204
task notify event word 210, 212
task preemption 46
Task preemption 51
task priorities 31
task priority 130
task ready list 139
tasks 17, 285
tasks blocked waiting for a queue 57
taskSCHEDULER_NOT_STARTED 259
taskSCHEDULER_RUNNING 259
taskSCHEDULER_SUSPENDED 259
task slices 46
task's local storage 258
task's stack size 37
task that has been suspended 132
task that is all ready to go 137
taskYIELD() 47, 48, 49, 63, 133
TCB 40
TCP/IP 31, 37
TCP/IP networking 19
temperature 120, 122
temporarily boosted 165
temporary boost 153
temporary priority boost 154, 165
TFT_eSPI 28, 186
TFT_eSPI.h 186
threshold 0 170
threshold 1 170
threshold interrupts 170
threshold status 176
throw-away code 289
throwie 114
tick interrupt 117
tick interrupts 143
tick period 47
ticks 204
TickType_t 20, 91
timeout 30, 236
timer 18, 143
timer callback 76, 78
timer command queue 77
timer daemon task 130
timer ID 77
timer instances 77
timer service task 77
timer tick 116
Time Sharing Option 17
time slice 18, 30, 44, 47, 48
time slices 131
tips 282
"Tmr Svc" task 77
trial measurements 172
trickle-down effect 144
troubleshooting 158
Troubleshooting 179, 187, 189, 272
TSO 17
TTGO ESP32 T-Display 169
TTGO ESP32 T-Display unit 28, 186
TTL to serial device 19
type option 289

U

udp_broadcast() 226
UDP broadcast server 227

UDP packets 228, 229
uint64_t 174
ulBitsToClearOnEntry 213
ulBitsToClearOnExit 213
ulTaskNotifyTake 204
ulTaskNotifyTake() 204, 205, 206, 207, 212
ultrasonic module 98
ultrasonic transducers 99
UML 285
unblocked 58, 59, 123, 152, 153, 177
unblocks 97
unequal execution time 150
unequal time slices 143
unit testing 261
unlock 154, 163
unlocked 154, 157
unlock too many times 163
Unreported errors 50
unused stack bytes 31
update the mailbox value 117
USB to serial interface 225
user provided callback 76
User_Setup.h 186
User_Setup_Select.h 186, 187
USE_SSD1306 99
uxBitsToSet 236
uxBitsToWait 224
uxBitsToWaitFor 224, 236
uxQueueMessagesWaiting() 72
uxQueueSpacesAvailable() 72
uxSemaphoreGetCount() 98
uxTaskGetNumberOfTasks 259
uxTaskGetStackHighWaterMark() 35

V

validation 19
variable length line of text 73
vEventGroupDelete() 241
virtual GPIO number 262
volatile 117
volatile attribute 117
volatile keyword 116
voluntary context switch 49
voluntary context switches 51
vSemaphoreDelete 155
vTaskDelay 75
vTaskDelay() 76, 90
vTaskDelay(10) 144
vTaskDelayUntil() 92, 95, 105
vTaskDelete() 42, 43
vTaskDelete(nullptr) 284
vTaskPrioritySet() 133
vTaskResume() 132, 138
vTaskSetThreadLocalStoragePointer 256
vTaskStartScheduler() 130
vTaskSuspend() 132
vTaskSuspendAll 259
vulnerable to exploits 291

W

wait for a notification 204
wait_ticks 55, 56
wakeup flag 206
watchdog 247, 249, 252
watchdog1.ino 246
watchdog is triggered 246
watchdog timer 243, 246, 251
watchdog timer reset call 244
watchdog timers 150, 243
Wemos Lolin ESP32 28, 169, 172, 177, 178
Wemos Lolin ESP32 dev board 28
WiFi 19, 31, 37, 98, 226, 229
WiFi credentials 225
WiFi facility 226
WiFi router 225, 226, 229
Windows Subsystem for Linux 289
wiring diagram 293
woken 176, 177, 197
WSL 229, 289

X

xClearCountOnExit 204, 205
xClearOnExit 224
xEventGroupClearBits 241
xEventGroupClearBitsFromISR() 241
xEventGroupCreate 223
xEventGroupCreateStatic 223
xEventGroupGetBits() 241
xEventGroupGetBitsFromISR() 241
xEventGroupSetBits 226

xEventGroupSetBits() 223, 224, 242
xEventGroupSetBitsFromISR 242
xEventGroupSetBitsFromISR() 223
xEventGroupSync() 236, 237
xEventGroupWaitBits() 223, 224, 242, 269
XGPIO 262, 263, 267, 268, 270, 271
XGPIO to pin mappings 262
xIndex 256
xPortGetCoreID() 36
xPortGetFreeHeapSize() 42
xQueueAddToSet 192
xQueueAddToSet() 192, 193, 194
xQueueCreate 57, 118
xQueueCreateSet() 198
xQueueCreateStatic 55
xQueueDelete 57
xQueueOverwrite 117
xQueueOverwrite() 118
xQueuePeek() 73, 118, 123
xQueueReceive 56, 57
xQueueReceive() 63, 118, 191, 193, 199
xQueueReset 57
xQueueReset() 68, 71, 73
xQueueSelectFromSet() 192, 198, 199, 202
xQueueSendFromISR() 176
xQueueSendFromISR() call 176
xQueueSendToBack 55
xQueueSendToBack () 56
xQueueSendToBackFromISR() 73
xQueueSendToFront() 56
xSemaphoreCreateMutex() 157
xSemaphoreCreateMutexStatic 154
xSemaphoreCreateMutex(void); 154
xSemaphoreCreateRecursiveMutex 164
xSemaphoreGive 154
xSemaphoreGive() 164
xSemaphoreGiveRecursive 164
xSemaphoreGiveRecursive() 164
xSemaphoreTake 154
xSemaphoreTake() 154, 164
xSemaphoreTakeRecursive 164
xSemaphoreTakeRecursive() 164
xTaskCallApplicationTaskHook() 260
xTaskCreate() 32, 132
xTaskCreatePinnedToCore 37
xTaskCreatePinnedToCore() 32, 37, 49
xTaskCreateStatic() 39, 40
xTaskCreateStaticPinnedToCore() 39
xTaskDelayUntil() 90, 91
xTaskGetCurrentTaskHandle() 44
xTaskGetSchedulerState 259
xTaskGetTickCount 90, 259
xTaskGetTickCount() 90
xTaskGetTickCountFromISR 259
xTaskNotify() 213
xTaskNotifyGive() 205, 207, 208
xTaskNotifyGiveFromISR() 206
xTaskNotifyWait 204
xTaskNotifyWait() 205, 212, 217
xTaskResumeAll 259, 260
xTimerChangePeriod() 81, 82
xTimerCreate() 79
xTimerReset() 81
xTimerSetTimerID() 78
xTimerStart() 79, 81
xWaitForAllBits 224

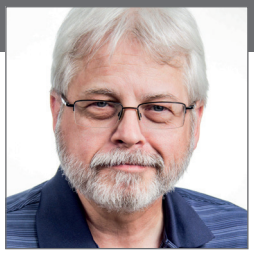
Y

yield 47, 149
yields 143

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

elektor