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.