EXPERT INSIGHT

Hands-On RTOS with Microcontrollers

Create high-performance, real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools



Second Edition

Jim Yuill | Penn Linder

packt

Hands-On RTOS with Microcontrollers

Second Edition

Create high-performance, real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools

Jim Yuill

Penn Linder

‹packt›

Hands-On RTOS with Microcontrollers

Second Edition

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Rohit Rajkumar

Relationship Lead: Kaustubh Manglurkar

Program Manager: Sandip Tadge

Content Engineer: Shazeen Iqbal

Technical Editor: Aditya Bharadwaj

Copy Editor: Safis Editing

Indexer: Manju Arasan

Proofreader: Shazeen Iqbal

Production Designer: Ajay Patule

Growth Lead: Lee Booth

First published: May 2020

Second edition: September 2025

Production reference: 1170925

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80323-772-5

www.packtpub.com

Dedicated to two friends and mentors,

in appreciation for their indispensable help, wisdom and good example:

Dr. E.C. Pasour (NCSU, ret.) and Colonel G.I. Wilson (USMC, ret.)

– Jim Yuill

Dedicated to those who forged the Embedded path:

You who fought for every last bit of RAM and ROM, armed with only Assembly language,

bound MCU reference manuals, LEDs and spare pins for debugging,

and a rotary phone for Tech Support.

May your passion and fortitude be the banner of the next generation.

– Penn Linder

Contributors

About the authors

Jim Yuill is a senior computer-systems engineer with 30 years of experience. He has worked in operating systems development, cybersecurity R&D, network systems-programming, and university teaching. He has a PhD in computer science, with a thesis in cybersecurity, which is highly cited.

Penn Linder is a Senior Electrical Engineer at IVEK Corporation, a manufacturer of industrial pumps and controllers. He has had a passion for embedded systems since taking his first embedded controllers class at Penn State University in 1995. He followed his passion by working for various companies that allowed him to design both hardware and software for products using embedded systems. Some of the products he has worked on include: lighting controls, medical devices, and industrial automation equipment. His early years were spent cranking out bare-metal assembly code for 8-bit MCUs. In his later years, he is writing software using C and FreeRTOS for 32-bit ARM Cortex-M processors.

I would like to extend my thanks to God who enables me to enjoy my labors and my family who enjoys the fruit of those labors with me.

About the reviewer

John Taylor has been an embedded developer for over 30 years. He has worked as a firmware engineer, technical lead, system engineer, software architect, and software development manager for companies such as Ingersoll Rand, Carrier, Allen-Bradley, Hitachi Telecom, Emerson, AMD, and several startup companies. He has developed firmware for products that include HVAC control systems, telecom SONET nodes, IoT devices, microcode for communication chips, and medical devices. He is the author of two books on embedded development, co-author of five US patents, and holds a bachelor's degree in mathematics and computer science.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers: `https://packt.link/embeddedsystems`.



Table of Contents

Preface xiii

Your Book Comes with Exclusive Perks - Here's How to Unlock Them xix

Part 1: Introduction to RTOSes, the Dev-Board, and Tools 1

Chapter 1: Introducing Real-Time Systems 3

What is "real-time" anyway? 4

Hardware and software roles in real-time systems 8

Defining RTOSes 11

Practical real-time systems 14

Choosing an RTOS 18

Summary 19

Chapter 2: Introducing the Development Board 21

Dev-board overview 21

Understanding the dev-board's components 24

Understanding the hardware documentation 32

Summary 36

Chapter 3: Introducing the Development-Tools 37

Using the book's instructions and example-programs 38

The IDE and the example-programs 39

The debugging tools: J-Link, Ozone and SystemView 49
Summary 62

Chapter 4: Understanding Super-Loops 63

Introducing super-loops 64
Improving the super-loop with interrupts 68
Adding a timer to the super-loop 72
Summary 82

Chapter 5: Implementing the Super-Loop 83

The super-loop example-program 84
Learning about system APIs and hardware features 97
How to develop the super-loop program 101
Summary 116

Part 2: The FreeRTOS Scheduler, Tasks, and Primary APIs 117

Chapter 6: Understanding RTOS Tasks 119

The FreeRTOS scheduler and tasks: a conceptual view 120
Choosing FreeRTOS as our RTOS 131
Hands-on with FreeRTOS's scheduler and tasks 132
Summary 148

Chapter 7: Running the FreeRTOS Scheduler 149

Creating tasks and starting the scheduler 150
Deleting tasks 158
Trying out the code 161
Understanding the scheduler 167
Task implementation and troubleshooting 176
Summary 184

Chapter 8: Protecting Data and Synchronizing Tasks 185

Running the chapter's example-programs 186
Semaphores 188
Priority inversion (how not to use semaphores) 206
Mutexes 213
Race conditions 220
Software timers 224
Summary 231
Further reading 231

Chapter 9: Intertask Communication 233

Understanding RTOS queues 234
Passing data through queues by value 240
Passing data through queues by reference 250
Task notifications 255
Summary 260
Further reading 260

Part 3: Using FreeRTOS to Build Systems 261

Chapter 10: Drivers and ISRs 263

Introducing the UART 264
Creating a polled UART driver 271
FreeRTOS-tasks vs. ISRs 278
Creating ISR-based drivers 285
Summary 300
Further reading 301

Chapter 11: More Efficient Drivers and ISRs 303

Creating a basic DMA-based driver 304
Creating an enhanced DMA-based driver 319
Choosing a driver model 334
Using third-party libraries and drivers 340
Summary 341
Further reading 342

Chapter 12: Sharing Hardware Peripherals among Tasks 343

Understanding how messages are sent over USB 344
Sending messages using a simple USB API 349
Creating a VCP driver, using a stream-buffer 354
Enhancing the VCP driver, for use by multiple tasks 362
Summary 382
Further reading 383

Chapter 13: Creating Loose Coupling with Queues 385

Creating a command queue 386
Reusing a queue definition for a new target 402
Understanding queues as interfaces 405
Summary 406
Further reading 406

Part 4: System Design and Troubleshooting 409

Chapter 14: FreeRTOS Memory Management 411

Understanding memory allocation 412
Static and dynamic allocation of FreeRTOS primitives 418
FreeRTOS heap implementations 423
Replacing malloc and free 428

Monitoring for stack and heap problems 429
Using a memory protection unit (MPU) 433
Summary 435
Further reading 435

Chapter 15: Multi-Processor and Multi-Core Systems 437

Introducing multi-core and multi-processor systems 438
Exploring multi-core systems 440
Exploring multi-processor systems 450
Exploring inter-processor communication 453
Choosing between multi-core and multi-processor systems 458
Summary 459
Further reading 459

Chapter 16: Troubleshooting Tips and Next Steps 461

General FreeRTOS debugging-techniques 462
Troubleshooting for the debugging tools 472
Next steps 484

Part 5: Reference Information and FreeRTOS Installation 485

Appendix A: Tools Quick-Reference 487

STM32CubeIDE 487
Ozone 488
SystemView 489

Appendix B: Reference Information 491

Chapters 2 and 3: Introducing the dev-board and tools 491
Chapter 4: Understanding super-loops 496
Chapter 5: Implementing the super-loop 496
Chapters 6 and 7: RTOS tasks and the FreeRTOS scheduler 497

Chapter 10: Drivers and ISRs **498**
Chapter 11: More efficient drivers and ISRs **498**
Chapter 12: Sharing hardware peripherals among tasks **499**
Chapter 14: FreeRTOS memory management **500**
Appendix C: FreeRTOS installation **501**

Appendix C: Creating FreeRTOS Projects, and Installing FreeRTOS **505**

Creating IDE-projects for use in learning FreeRTOS **505**
Copying and renaming projects (cloning) **506**
Installing FreeRTOS and SystemView **509**

Other Books You May Enjoy **533**

Index **537**

Preface

This second-edition hands-on guide will provide you with the functional knowledge needed for using a **real-time operating system** (**RTOS**) in embedded systems, and in particular, systems that run on a **microcontroller unit** (**MCU**). If you're interested in learning how to implement such systems through hands-on examples and using actual hardware, you're in the right place!

This book focuses on FreeRTOS, a market-leading and open-source RTOS. Its features and application are brought to life with more than 25 example-programs.

The book explains how FreeRTOS works, including its primary features and APIs. The explanations are accompanied by example programs, and they are described line-by-line. The book also shows how FreeRTOS can be used to create real-world systems, such as drivers for MCU peripherals.

The example programs run on an ST **development board** (**dev-board**), which can be purchased separately (STM32 Nucleo-F767ZI). A set of development tools is presented, and these tools are used when working with the example programs. The tools include the IDE STM32CubeIDE and SEGGER's Ozone and SystemView debugging tools. SystemView is especially valuable for its FreeRTOS-related features, including its graphical display of FreeRTOS's multi-tasking.

You will gain hands-on experience with FreeRTOS and the development tools. Using the dev-board and tools, you can run the example programs, analyze their behavior, and experiment with them by making modifications. Also, instructions are provided for installing FreeRTOS on the dev-board, so you can develop your own programs from scratch.

Engineers who are already familiar with such dev-boards and tools may want to just read the book and not use the dev-board and tools. The book shows the example-programs' code, and tool use is shown in the figures.

Who this book is for

This book is intended for embedded-systems engineers. It's also intended for programmers without embedded systems experience, but who have a good understanding of system architecture, such as systems programmers. A basic understanding of the C programming language will be helpful.

The book is written for a wide audience, ranging from college upperclassmen to senior engineers. Tutorial information is provided for readers with less experience. Readers with more experience can skip the tutorials; the tutorials make up a relatively small part of the book.

What this book covers

This book has 16 chapters, and they are divided into 4 parts. There are also 3 appendices.

Part 1: Introduction to RTOSes, the Dev-Board, and Tools

Chapter 1: RTOSes are described, including when to use them.

Chapters 2-3: The book is accompanied by a dev-board (purchased separately), development tools, and example programs. They are described, as well as how to use them. Tutorial information is provided for those who are new to embedded-systems hardware and the development tools.

Chapters 4-5: Super-loops are presented. They show the need for an RTOS and its benefits. A tutorial is provided on how to create a super-loop program. The tutorial is intended for programmers who are new to embedded systems.

Part 2: The FreeRTOS Scheduler, Tasks, and Primary APIs

Chapters 6-7: FreeRTOS's core components are presented: the scheduler and tasks.

Chapters 8-9: FreeRTOS's APIs are presented for task synchronization and communication, such as semaphores and queues.

Part 3: Using FreeRTOS to Build Systems

Chapters 10-12: These chapters show how FreeRTOS's features can be used in real-world systems, and in particular, to create drivers for MCU peripherals.

Chapter 13: This chapter shows how tasks can be used to process serial data-streams, and specifically, data from the MCU's USB controller.

Part 4: System Design and Troubleshooting

Chapters 14-15: FreeRTOS memory management is presented, as well as FreeRTOS use with multi-processor and multi-core systems.

Chapter 16: Troubleshooting techniques are shown for FreeRTOS-based apps and for the development tools.

Appendices: Reference Information and FreeRTOS Installation

Appendix A: A "cheat sheet" is provided for the development tools.

Appendix B: Reference documents are listed for the development tools and the hardware, for example, user guides.

Appendix C: This appendix shows how to install FreeRTOS on the dev-board.

To get the most out of this book

For many readers, hands-on use of the book's example programs will be invaluable for gaining skills. This includes using the debugging tools to observe and analyze the programs' behavior. Hands-on use requires having the dev-board, installing the development tools, and downloading the example programs.

Conversely, for experienced engineers, it may be most efficient to simply read the book and not run the example programs.

The book explains the example programs' primary parts. As you read the book, it's also helpful to have the IDE available, to view the full code-listings and explore the example programs in more detail.

To master the material in this book, we recommend two approaches:

- Experiment with modifying the example programs.
- Create your own FreeRTOS-based programs from scratch, using the FreeRTOS features from the book. *Appendix C* shows how to create an IDE-project for the dev-board and install FreeRTOS.

Download the example code files

The example-programs for the book are hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-RTOS-with-Microcontrollers-Second-Edition`. Instructions for downloading, installing, and using these programs are provided in *chapter 3*.

We also have code bundles from our catalog of books and videos available at `https://github.com/PacktPublishing`. Check them out!

The Hardware and Development Tools

To run the example-programs, an STM32 Nucleo-F767ZI dev-board is needed. In the future, if additional hardware is supported, it will be described in the book's GitHub repository:

`https://github.com/PacktPublishing/Hands-On-RTOS-with-Microcontrollers-Second-Edition`.

The development-tools and how to use them are presented in *Chapter 3*, *Introducing the Development Tools*. In writing the book, the Windows 11 versions of the tools were used. The tools are also available for macOS and Linux, and those versions are expected to work adequately with the book, though we have not fully tested them.

The example-programs can be downloaded from the book's GitHub repository. This is also described in *chapter 3*.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781803237725`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "both `globalVar` and `staticVar` are located in static memory"

A block of code is set as follows:

```
void vApplicationStackOverflowHook( void )
{
    __disable_irq();
    while(1);
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in highlighted:

```
void vcpTransmitComplete( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    // Give the semaphore, to indicate the transmit completed
    xSemaphoreGiveFromISR( vcpTransmitCompleteSemaphore,
                           &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Any command-line input or output is written as follows:

```
Disabled output of control characters
SEGGER Ozone - The J-Link Debugger V3.38f
J-Link software found at: C:/Program Files/SEGGER/Ozone V3.38f/JLink_x64.dll
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Open the **STM32CubeIDE**, and the example-programs' workspace."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`

Share your thoughts

Once you've read *Hands-On RTOS with Microcontrollers, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Your Book Comes with Exclusive Perks - Here's How to Unlock Them

Unlock this book's exclusive benefits now

Scan this QR code or go to `https://packtpub.com/unlock`, then search for this book by name. Ensure it's the correct edition.

UNLOCK NOW

Note: *Keep your purchase invoice ready before you start.*



Enhanced reading experience with our Next-gen Reader:

- **Multi-device progress sync**: Learn from any device with seamless progress sync.
- **Highlighting and notetaking**: Turn your reading into lasting knowledge.
- **Bookmarking**: Revisit your most important learnings anytime.
- **Dark mode**: Focus with minimal eye strain by switching to dark or sepia mode.

Learn smarter using our AI assistant (Beta):

- **Summarize it**: Summarize key sections or an entire chapter.
- **AI code explainers**: In the next-gen Packt Reader, click the **Explain** button above each code block for AI-powered code explanations.



Note: The AI assistant is part of next-gen Packt Reader and is still in beta.

Learn anytime, anywhere:

Access your content offline with DRM-free PDF and ePub versions—compatible with your favorite e-readers.

Unlock Your Book's Exclusive Benefits

Your copy of this book comes with the following exclusive benefits:

Next-gen Packt Reader

AI assistant (beta)

DRM-free PDF/ePub downloads

Use the following guide to unlock them if you haven't already. The process takes just a few minutes and needs to be done only once.

How to unlock these benefits in three easy steps

Step 1

Keep your purchase invoice for this book ready, as you'll need it in *Step 3*. If you received a physical invoice, scan it on your phone and have it ready as either a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/unlock-benefits/help`.



Note: Did you buy this book directly from Packt? You don't need an invoice. After completing Step 2, you can jump straight to your exclusive content.

Step 2

Scan this QR code or go to `https://packtpub.com/unlock`.

On the page that opens (which will look similar to Figure 0.1 if you're on desktop), search for this book by name. Make sure you select the correct edition.

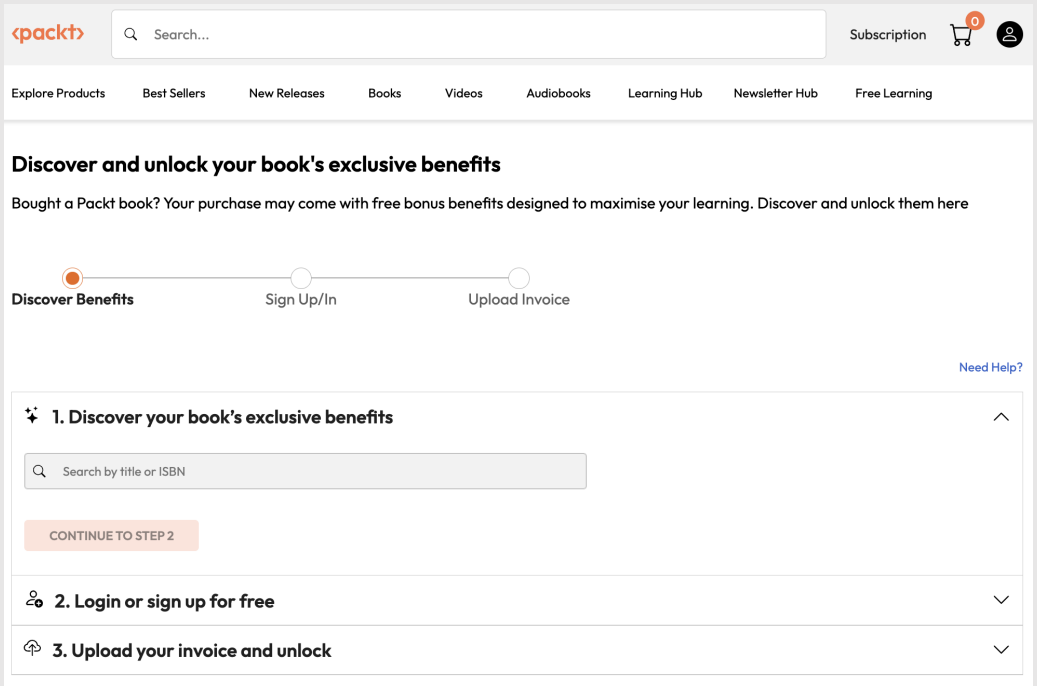


Figure 0.1: Packt unlock landing page on desktop

Step 3

Sign in to your Packt account or create a new one for free. Once you're logged in, upload your invoice. It can be in PDF, PNG, or JPG format and must be no larger than 10 MB. Follow the rest of the instructions on the screen to complete the process.

Need help?

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. The following QR code will take you to the help page directly:



Note: If you are still facing issues, reach out to `customercare@packt.com`.

Part 1

Introduction to RTOSes, the Dev-Board, and Tools

The first part of the book introduces real-time operating systems (RTOSes). Also, it introduces the development board (dev-board) that we'll be using to run the book's example programs. We'll see how to download the example programs and how to install and run the development tools that are used with the example programs. These tools include an IDE and two SEGGER debugging tools. In addition, we'll present super-loops, a common program structure used in bare-metal programming. Super-loops provide a frame of reference for understanding the value of RTOSes, and you will see how RTOSes provide capabilities far beyond those of standard super-loops. Part 1 includes tutorial sections for programmers who are new to embedded systems. Experienced embedded engineers can skim or skip the tutorial sections. By the end of Part 1, you'll have the background needed for hands-on learning about RTOSes, and specifically, FreeRTOS.

Part 1 includes the following chapters:

- *Chapter 1*, *Introducing Real-Time Systems*
- *Chapter 2*, *Introducing the Development Board*
- *Chapter 3*, *Introducing the Development Tools*
- *Chapter 4*, *Understanding Super-Loops*
- *Chapter 5*, *Implementing the Super-Loop*

1

Introducing Real-Time Systems

Real-time systems come in a wide variety of implementations and use cases. This book focuses on how to use a **real-time operating system** (**RTOS**) to create real-time applications on a **microcontroller unit** (**MCU**).

In this chapter, we'll start with an overview of what an RTOS is and get an idea of the wide range of systems that can have real-time requirements. From there, we'll look at some of the different ways of achieving real-time performance, along with an overview of the types of systems (such as hardware, firmware, and software) that may be used. We'll wrap up by discussing how to choose an RTOS, when it is advisable to use an RTOS in an MCU application, and when it might not be necessary at all.

For readers who are not familiar with MCUs, tutorial information on MCUs can be found in *Chapter 2*, *Introducing the Development Board*.

In a nutshell, we will cover the following topics in this chapter:

- What is "real-time" anyway?
- Hardware and software roles in real-time systems
- Defining RTOSes
- Practical real-time systems
- Choosing an RTOS

What is "real-time" anyway?

Real-time systems are systems that have timing requirements. A real-time system is considered to **fail** when it doesn't meet a timing requirement. How a **timing failure** is defined (and the consequences of a failed system) can vary widely. It is extremely important to realize that **real-time requirements** can vary widely, both in the **speed** of the timing requirement and the **severity** of consequences if the required real-time deadlines are not met.

Every real-time system is constrained in its ability to respond to events and/or signals. A system's ability to respond to events and/or signals is referred to as the **system's bandwidth**. For example, when designing a signal acquisition chain, the system's bandwidth for each filter stage must be wide enough to pass through the desired signal in order to maintain the integrity of the signal. If the system's bandwidth for the filters is not properly designed, the system will fail to provide the desired real-time response to certain input signals.

The system's hardware and software both have **limited bandwidth**. All software executes on hardware that has limited resources. For example, the clock frequency of an MCU limits how many instructions it can execute in a given period of time. The rate of instruction execution is also limited by the MCU architecture, which determines how many clock cycles are required to execute each instruction. The ability of the software to respond to events in real time is affected by the algorithms used, the **operating system** (**OS**) used (if any), and the effective use of the OS's features, such as multitasking and timing features.

The limitations of the hardware and software make real-time design both challenging and fun. Challenging, because the designer often has to look into the nitty-gritty of the MCU architecture, including the hardware peripheral registers and the assembly code instructions. But fun, from the satisfaction of understanding the intricacies of how the hardware works, and from being able to write software that takes advantage of it.

It is both a science and an art to balance these limitations and to understand the trade-offs between hardware and software in order to get to the end goal of a *working* real-time system.

The range of timing requirements

To illustrate the range of timing requirements that can be encountered, let's consider a few different systems that acquire readings from **analog-to-digital converters (ADCs)**.

Let's first look at a lower-bandwidth control system. The control system is part of a **soldering iron**, and the control system maintains the tip's temperature (as seen in *Figure 1.1*). The parts of the system we're concerned with are the MCU, ADC, sensor, and heater.

The MCU is the brain of the control system. It is responsible for the following:

- Taking readings from a temperature sensor via the ADC
- Running a closed-loop control algorithm (to maintain a constant temperature at the soldering iron tip)
- Adjusting the output of the heater as needed

These can be seen in the following figure:

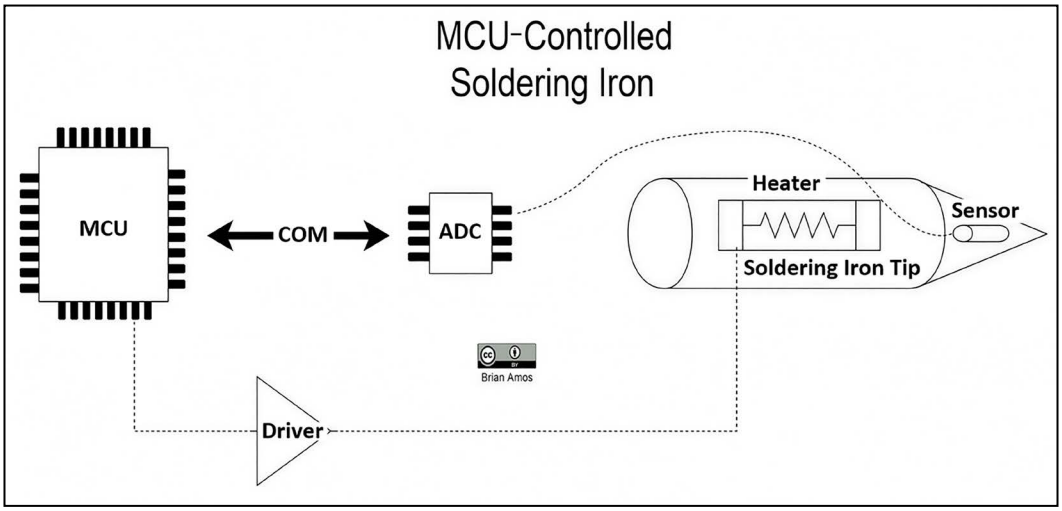


Figure 1.1: MCU-controlled soldering iron

A typical soldering iron would have pretty low bandwidth needs. For example, since the temperature of the tip doesn't change very quickly, the MCU may only need to acquire 50 ADC samples per second (50 Hz). The control algorithm responsible for adjusting the heater (to maintain a constant temperature) may run at an even slower pace, perhaps 5 times per second (5 Hz).

While we are here, let's take a quick look at the timing requirements of the ADC. Doing so will give us a feel for some of the time constraints we deal with when designing real-time systems. *Figure 1.2* shows a common interface between an MCU and an ADC:

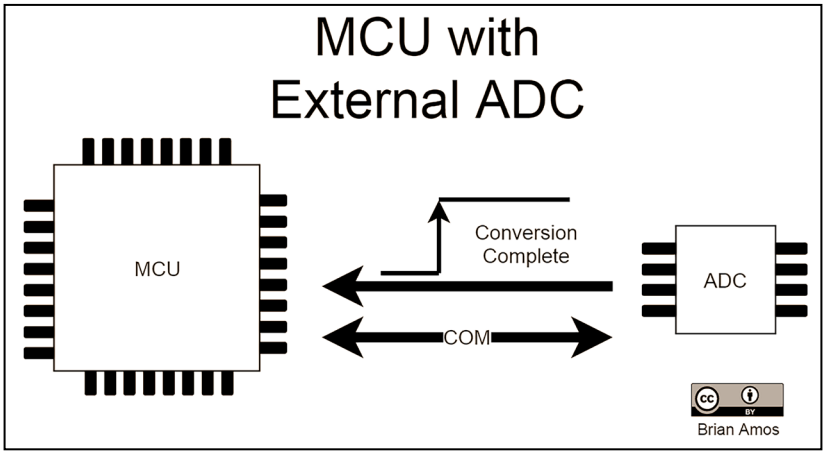


Figure 1.2: MCU with external ADC

The ADC asserts a signal to inform the MCU that the ADC has completed a conversion of the analog signal and that a digital value is ready for the MCU to read. The MCU must read the converted digital value, using the communication channel, before the next conversion takes place, or the result will be overwritten. The MCU has up to 20 ms to transfer the data from the ADC to internal memory before a new reading needs to be taken (as seen in *Figure 1.3* below). The MCU also needs to be running the control algorithm to calculate the updated values for the heater output at 5 Hz (200 ms). Both of these cases (although not particularly fast) are examples of real-time requirements.

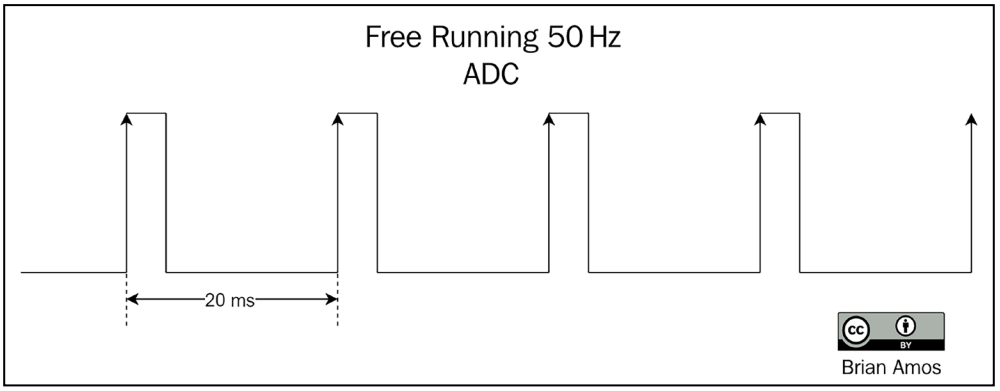


Figure 1.3: Free-running 50-Hz ADC

Now that we have looked at a lower-bandwidth control system, let's consider that other control systems may require much higher bandwidth. For example, a network analyzer or oscilloscope may require reading an ADC at a rate of tens of GHz! The raw ADC readings will likely be converted into the frequency domain and graphically displayed on a high-resolution front panel dozens of times a second. To function properly, such a system requires significant data-processing bandwidth, and it must adhere to extremely tight timing requirements.

Somewhere in the middle of the possible bandwidth requirements, you'll find systems such as closed-loop motion controllers, which will typically need to execute their **proportional–integral–derivative** (**PID**) control loops between hundreds of Hz to tens of kHz in order to provide stability in a fast-moving system. So, *how fast* is real-time? Well, as you can see from the ADC examples alone, it depends on the *requirements*.

In some of the previous cases, such as the oscilloscope or soldering iron, failure to meet a timing requirement results in poor performance or incorrect data being reported. In the case of the soldering iron, this might be poor temperature control (which could cause damage to components). For the oscilloscope, missing deadlines could cause erroneous readings, which is a failure. Too many inaccurate readings will cause the device to be viewed as unreliable, and sales will decline—all because a real-time requirement wasn't being met consistently.

In other systems, such as the flight control of an **unmanned aerial vehicle** (**UAV**) or motion control in industrial process control, failing to run the control algorithm in a timely manner could result in something more physically catastrophic, such as a crash. In this case, the consequences are potentially life-threatening.

Thankfully, there are steps that can be taken to avoid all of these failure scenarios.

The ways of guaranteeing real-time behavior

One of the easiest ways to ensure that a system does what it is meant to do is to make sure it is as simple as possible while still meeting the requirements (some call this the **KISS** principle—**Keep It Simple, Stupid!**). This means resisting the urge to over-complicate a simple task. If a toaster is meant to toast a slice of bread, don't put a display on it and make it tell you the weather too; just have it turn on a heating element for the right amount of time. This simple task has been accomplished for years without requiring any code or programmable devices whatsoever.

As programmers, when we encounter a problem, we tend to reach for the nearest MCU and start coding. However, some functions of a product are best handled without code at all (this is especially pertinent if a product has electro-mechanical components). A car window doesn't really need an MCU with a polling loop to run, turning on motors through drivers and watching sensors for feedback to shut them off. This task can actually be handled by a few mechanical switches and diodes. If a feedback-reporting mechanism is required for a given system—such as an error that needs to be asserted in the case of a stuck window—then there may be no choice but to use a more complex solution. However, our goal as engineers should always be the same—solve the problem as *simply* as possible, without adding additional complexity.

If a problem can be solved by hardware alone, then explore that possibility with the team first before breaking out the MCU. If a problem can be handled by using a simple `while` loop to perform some polling of the sensor status, then simply poll the sensor for the status; there may be no need to start coding **interrupt service routines** (**ISRs**) (we will discuss these later in the book). If the functionality of the device is single-purpose, there are many cases where a full-blown RTOS can simply get in the way—so don't use one!

Hardware and software roles in real-time systems

There are many different trade-offs when using hardware and software to achieve real-time behavior in a system. The following sections discuss the various ways hardware and software are deployed in real-time systems you might encounter. Also, note that it is possible to have combinations of the following systems working together as subsystems. These different subsystems can occur at the product, board, or even chip level (this approach is discussed in *Chapter 15*, *Multi-Processor and Multi-Core Systems*).

Hardware-only

The original real-time system is **hardware-only**, and it is often the best solution when there are extremely tight tolerances and/or fast timing requirements. It can be implemented with discrete digital logic, analog components, programmable logic, or an **application-specific integrated component** (**ASIC**). Other options for the hardware include **programmable logic devices** (**PLDs**), **complex programmable logic devices** (**CPLDs**), and **field-programmable gate arrays** (**FPGAs**). Hardware-based real-time systems can cover anything from analog filters, closed-loop control, and simple state machines to complex video codecs. When implemented with power saving in mind, ASICs can be made to consume less power than an MCU-based solution. In general, hardware has the advantage of performing operations *in parallel* and *instantly* (this is, of course, an over-simplification), as opposed to a single-core MCU, which only gives the illusion of parallel processing.

The downsides for real-time hardware development generally include the following:

- The inflexibility of non-programmable devices
- The expertise required is generally less commonly available than software/firmware developers
- The cost of full-featured programmable devices (for example, large FPGAs)
- The high cost of developing a custom ASIC

Bare-metal firmware

Bare-metal firmware is considered (for our purposes) to be any firmware that doesn't use a preexisting RTOS. Instead, the designer must write firmware for the features that an RTOS would typically provide. Some engineers take this a step further, arguing that true bare-metal firmware can't use any preexisting libraries (such as vendor-supplied hardware-abstraction libraries)—there is some merit to this view as well. A bare-metal implementation has the advantage that the user's code has *total* control of *all* aspects of the hardware. The only way for the main-loop code execution to be interrupted is if an interrupt occurs, giving control over to an **ISR**. In this case, the only way for anything else to take control of the CPU is for the existing ISR to finish or for a higher-priority interrupt to occur.

Bare-metal firmware implementations excel when there is a small number of relatively simple tasks to perform—or one monolithic task. If the firmware is kept focused and best practices are followed, deterministic performance is generally easy to measure and guarantee due to the relatively small number of interactions between ISRs (or in some cases, a lack of ISRs).

In some extreme cases for heavily loaded MCUs (or MCUs that are highly constrained in ROM/RAM), bare-metal is the only option. Back when bare-metal development first started, the MCUs were highly constrained, and programming bare-metal was a necessity. However, as the silicon industry has matured, more and more memory fits inside a tiny MCU. There are now many devices available, with plenty of memory, and at a low cost, which makes it unnecessary to limit yourself to bare-metal programming. Often, it makes sense to use an RTOS, which we will explore next.

RTOS-based firmware

As bare-metal implementations become more elaborate when dealing with events asynchronously, they start to overlap with the functionality provided by an RTOS. An important consideration to keep in mind is that by using an RTOS—rather than attempting to roll your own thread-safe system—you automatically benefit from all of the testing the RTOS provider has put in.

You'll also have the opportunity to use code that has the power of hindsight behind it—many of the RTOSes available today have been around for several years. The authors have adapted and added functionality the entire time to make the RTOSes robust and flexible for different applications. Using one of these mature RTOSes provides a high-quality starting point for your application.

RTOS-based firmware differs from bare-metal firmware in that it runs on top of a scheduling kernel on an MCU. For RTOS-based firmware, the term **firmware** signifies a type of RTOS that is very limited in regard to its features – typically it operates out of limited flash memory (e.g., 64KB, 256KB, etc.), and it provides a scheduling kernel for use by tasks. It is like bare-metal firmware in that the programmer often has to be very aware of the hardware peripherals and registers offered by the MCU. However, the RTOS often abstracts away some of those details and also provides timing mechanisms that the application firmware can take advantage of.

With an RTOS, having a scheduler and some RTOS primitives allows tasks to operate under the illusion that they have the processor to themselves (discussed in detail in *Chapter 6*, *Understanding RTOS Tasks*). Using an RTOS enables the system to remain responsive to the most important events while performing other complex tasks *in the background*.

There are a few downsides to having several tasks running on the processor. Sharing data between tasks requires the proper use of synchronization methods to avoid problems such as **deadlock** and **priority inversion**. Later in the book, we will discuss some of these synchronization methods, although it adds complexity to the code. Interrupts will generally use task signaling to take care of the interrupt as quickly as possible and defer as much processing to a task as possible. If handled properly, this solution is excellent for keeping complex systems responsive, despite many complex interactions.

RTOS-based software

RTOS-based software differs from RTOS-based firmware in that this software runs on a more full-featured RTOS. The features are similar to those found in a general-purpose OS and may include things such as hardware drivers, bootloaders, applications, filesystems, etc. In fact, some general-purpose OSes offer real-time variants (e.g., RTLinux).

These RTOSes typically require significantly more memory (e.g., 1 MB or more) and processors with specific features. For example, the RTOS may make use of the processor's **memory management unit** (**MMU**). Also, the RTOS may provide features such as application loading and a defined driver framework. The software applications for an RTOS can be highly complex, requiring many different interactions between various internal and external systems. The advantage of using this type of RTOS is all of the capabilities that come along with it—both hardware and software.

In addition, on the hardware side, a more complex processor is often used to provide additional event-handling capacity (bandwidth) to the application. For example, there may be multiple CPU cores available running at higher clock rates. There can be GBs of RAM and persistent memory available. Adding peripheral hardware can be as simple as the addition of a card (provided there are pre-existing drivers).

On the software side, there is a plethora of open-source and vendor-proprietary solutions for networking stacks, UI development, file handling, and so on. The availability of these solutions means that certain features of the end product will require less development time to implement. Underneath all of these capabilities and options, the kernel is still implemented in such a way that the critical tasks won't be blocked for an indefinite period of time, which is possible with a general-purpose OS. Because of this, getting deterministic performance is still within reach, just like with RTOS-based firmware.

Carefully crafted OS software

Similar to RTOS-based software, a standard OS has all of the libraries and features a developer could ask for. What's missing, however, is a strict focus on meeting timing requirements. Generally speaking, systems implemented with a traditional OS are going to have much less deterministic behavior (and none that can be truly counted on in a safety-critical situation). For example, how many times have you sat at your PC and had a program indicate "not responding"? Now imagine if your PC also happened to be controlling the timer on your toaster – you might end up with some burnt toast!

If there is a lax real-time requirement without catastrophic consequences if a wishy-washy deadline isn't met on time, a standard OS can be made to work, as long as care is taken in choosing what software stacks are running and their resource use is kept in check. The Linux kernel with `PREEMPT_RT` patches is a good example of this type of RTOS.

So, now that several of the options for achieving a real-time system with hardware and software have been laid out, it's time to define exactly what we mean when we refer to an RTOS, specifically an MCU-based RTOS.

Defining RTOSes

An RTOS is a version of an OS that is carefully designed to offer real-time mechanisms to programmers. General-purpose OSes (such as Windows, Linux, and macOS) were created to provide a consistent programming environment that abstracts away the underlying hardware to make it easier to write and maintain computer programs. They provide application programmers with many different **primitives,** such as threads and mutexes.

Threads are sections of software that have their own stack and operating context. Threads are like tasks in that they run independently from each other. **Mutexes** are primitives offered by OSes. Mutexes provide a means for threads to synchronize their processing.

Threads (or **tasks**) and mutexes can be used by application software to create more complex behavior. For example, it is possible to create a multi-threaded program that provides protected access to shared data:

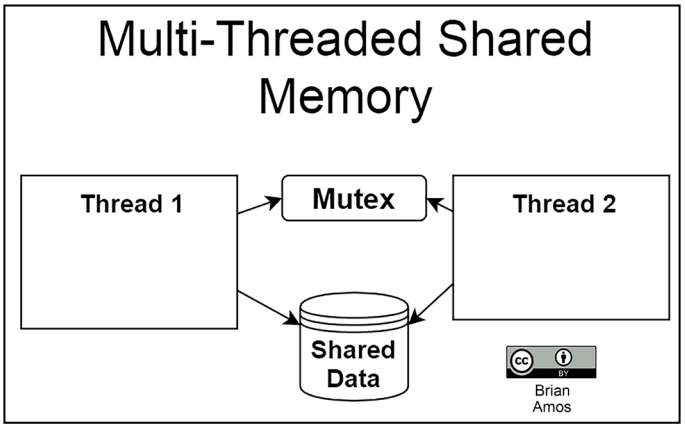


Figure 1.4: Multi-threaded shared memory

The preceding application doesn't *implement* thread and mutex primitives, it only makes use of them. The actual implementations of threads and mutexes are handled by the OS. This has a few advantages:

- The application code is less complex
- It is easier to understand—the same primitives are used regardless of the programmer, making it easier to understand code created by different people
- There is better hardware **portability**—with the proper precautions, the code can be run on any hardware supported by the OS without modification

In the preceding example, a mutex is used to ensure that only one thread can access the **shared data** at any given time. In the case of a general-purpose OS, each thread will happily wait indefinitely for the mutex to become available, before moving on to access the shared data. This is where an RTOS diverges from a general-purpose OS. In an RTOS, all blocking system calls are **time-bound**; instead of waiting for the mutex indefinitely, an RTOS allows a maximum delay to be specified. For example, with a general-purpose OS, if `Thread 1` attempts to acquire `Mutex` and still doesn't have it after 100 ms, or even 100 seconds, it will continue waiting for `Mutex` to become available—even if it takes *forever*.

In an RTOS implementation, the maximum amount of time to wait for `Mutex` to become available is specified by the programmer. The programmer could specify that `Thread 1` must acquire `Mutex` within 100 ms. If `Thread 1` still hasn't received `Mutex` after 101 ms, `Thread 1` will receive a notification that `Mutex` hasn't been acquired by the specified time. This is called a **timeout,** and it is specified to help create a deterministic system.

An OS that provides a way of executing a given piece of code, within specified timing requirements, can be considered a **real-time OS**. This definition of an RTOS covers a fairly large number of systems.

There are a couple of characteristics that tend to differentiate applications that use an RTOS: how often *not* meeting a real-time deadline is acceptable, and the *severity* of not meeting a real-time deadline. The different ranges of applications that use an RTOS are usually lumped into three categories—hard, firm, and soft real-time systems.



Don't get too hung up on the differences between *hard*, firm, and soft real-time systems. The definitions for these terms don't even have unanimous agreement from within our industry. What *does* matter is that you know your system's requirements and design a solution to meet them!

Hard real-time systems

A **hard real-time system** must meet its deadline 100% of the time. If the system does not meet a deadline, then it is considered to have **failed**. This doesn't necessarily mean a failure will hurt someone if it occurs in a hard real-time system—only that the system *has* failed if it misses a single deadline.

The **severity of a failure** is generally deemed **safety-critical** if a failure will cause the loss of life or significant property. There are also hard real-time systems that have nothing to do with safety.

Some examples of hard real-time systems can be found in medical devices, such as pacemakers and control systems with extremely tightly controlled parameters. In the case of a pacemaker, if the pacemaker misses a deadline to administer an electrical pulse at the right moment in time, it might kill the patient (this is why pacemakers are defined as safety-critical systems).

In contrast, if a motion control system on a **computer numerical control (CNC)** milling machine doesn't react to a command in time, it might plunge a tool into the wrong part of the item being machined, ruining it. In these cases that we have mentioned, one failure caused a loss of life, while the other turned some metal into scrap—but both were failures caused by a single missed deadline.

Firm real-time systems

As opposed to hard real-time systems, **firm real-time systems** need to hit their deadlines *nearly* all of the time. If video and audio lose synchronization momentarily, it probably won't be considered a system failure, but will likely upset the consumer of the video.

In most control systems (similar to the soldering iron in a previous example), if a few samples are read slightly outside of their specified time, it is unlikely the system will be destroyed. If a control system has an ADC that automatically takes a new sample, and the MCU doesn't read the new sample in time, it will be overwritten by a new one. This can occur occasionally, but if it happens too often, the temperature stability will be ruined. However, other systems cannot tolerate missed samples. Motor control loops, for example, might lose control of the motor if even a few samples are missed. This highlights the need for the programmer to understand the real-time requirements of the system and to design accordingly.

Soft real-time systems

Soft real-time systems are the most lax when it comes to how often the system must meet its deadlines. These systems often offer only a *best-effort* promise to keep deadlines. These real-time systems don't consider it a *failure* when their deadlines are not met, though the user of the system might consider it an *annoyance*.

Cruise control in a car is a good example of a soft real-time system because there are no hard specifications or expectations of it. Drivers typically don't expect their speed to converge to within +/- *X* mph/kph of the set speed, within N seconds. They expect that given *reasonable* circumstances, such as no large hills, the control system will eventually get them *close* to their desired speed *most of the time*. However, they might still be annoyed when they have to step on the gas to get the car to climb that large hill!

Next, we'll look at the various types of RTOSes that are currently available, and the RTOS used in this book.

Practical real-time systems

In practice, real-time systems often have a combination of firm, hard, and soft real-time requirements. For example, the soldering iron example may have hard real-time requirements for a thermal run-away safety monitor, firm real-time requirements for the temperature measurement and control loop, and soft real-time requirements for updating the user display. Achieving this combination of real-time requirements is made easier by using an RTOS, but it requires careful attention to these constraints when designing the system.

The range of RTOSes

There are many RTOSes on the market. They range widely in their functionality, and in what processor architecture and size they're best suited to. On the smaller side are 8-bit to 32-bit MCU-focused RTOSes, including FreeRTOS, Keil RTX, Micrium µC/OS, ThreadX, and many more. This class of RTOS is suitable for use on microcontrollers, and a compact real-time kernel is provided as the most basic offering. When moving from MCUs to 32-bit and 64-bit application processors, there are RTOSes such as Wind River VxWorks, Wind River Linux, Green Hills' Integrity OS, and even Linux with `PREEMPT_RT` kernel extensions. These full-blown OSes offer a large selection of software, providing solutions for both real-time scheduling requirements as well as general computing tasks. Even with the OSes we've just rattled off, we've only scratched the surface of what's available. There are free and paid solutions (some costing well over $10,000) at all levels of RTOSes, big and small.

So, why would you choose to pay for a solution when there is something available for free? The main differentiating factors between freely available RTOS solutions and paid solutions are safety approvals, middleware, and customer support. Because an RTOS provides a highly deterministic execution environment, they are often used in complex **safety-critical** applications. These systems require deterministic operation because they must behave in a predictable way all the time. Guaranteeing that the code responds to events within a fixed amount of time is a significant step toward ensuring they behave consistently. Most of these safety-critical applications are regulated and have their own sets of governing bodies and standards, such as *DO-178B* and *DO-178C* for aircraft or *IEC 61508 SIL 3* and *ISO 26262* ASILD for industrial applications. To make safety-critical certifications more affordable, designers typically use one of two approaches. One approach is to keep code for these systems extremely simple, and not to use an RTOS. This makes it possible to prove mathematically that the system will function consistently and nothing can go wrong. The other approach is to turn to a commercial RTOS solution, which has been through the certification process, as a starting point. For example, WITTENSTEIN SafeRTOS is a derivative of FreeRTOS that carries approvals for industrial, medical, and automotive use.

Middleware can also be an extremely important component in complex systems. **Middleware** is code that runs between the **user code** (code that *you*, the application programmer, write) and lower layers, such as the RTOS or bare-metal (no RTOS). A value proposition of paid solutions is that the ecosystem offers a suite of pre-integrated high-quality middleware (such as filesystems, networking stacks, GUI frameworks, industrial protocols, and so on), and it minimizes the amount of development required and reduces overall project risk.

The reason for using middleware, rather than *rolling your own*, is to reduce the amount of original code being written by an in-house development team. This reduces both the risk and the total time spent by the team—so it can be a worthwhile investment, depending on factors such as project complexity and schedule requirements.

Paid solutions will also typically come with some level of customer support directly from the firmware vendor. Engineers are expensive to hire and keep on staff. There's nothing a manager dreads more than walking into a room full of engineers who are puzzling over their tools, rather than working on the *real* problems that need to be solved. Having expert help that is an email or phone call away., can increase a team's productivity dramatically, which leads to a shorter turnaround and a happier workplace for everyone.



FreeRTOS has both paid support and training options, as well as paid middleware solutions that can be integrated. However, open-source and other free middleware components are also available, some of which will be discussed later in this book.

The RTOS used in this book

With all of the options available, you might be wondering: why is it that this book is only covering one RTOS on a single model of MCU? There are a few reasons, one being that most of the concepts we'll cover are applicable to nearly any RTOS available, in the same way that good coding habits transcend the language in which you happen to be coding. By focusing on a single implementation of an RTOS with a single MCU, we'll be able to dive into topics in more depth than would have been possible if all of the alternatives were also attempted to be discussed.

FreeRTOS is one of the most popular RTOS implementations for MCUs, and it is very widely available. It has been around for over 15 years and has been ported to dozens of platforms. If you've ever spoken to a true low-level embedded systems engineer who is familiar with RTOS programming, they've certainly heard of FreeRTOS and have likely used it at least once. By focusing our attention on FreeRTOS, you'll be well positioned to quickly migrate your knowledge of FreeRTOS to other hardware or transition to another RTOS, if the situation calls for it.



The other reason we're using FreeRTOS? Well, it's FREE! FreeRTOS is distributed under the MIT license. See `https://www.freertos.org/a00114.html` for more details on licensing and other FreeRTOS derivatives, such as SafeRTOS and OpenRTOS.

The following is a diagram showing where FreeRTOS sits in a typical ARM **firmware stack**. **Stack** refers to all of the different *layers* of firmware components that make up the system and how they are *stacked* on top of one another. A **user** in this context refers to the programmer using FreeRTOS (rather than the end user of the embedded system):

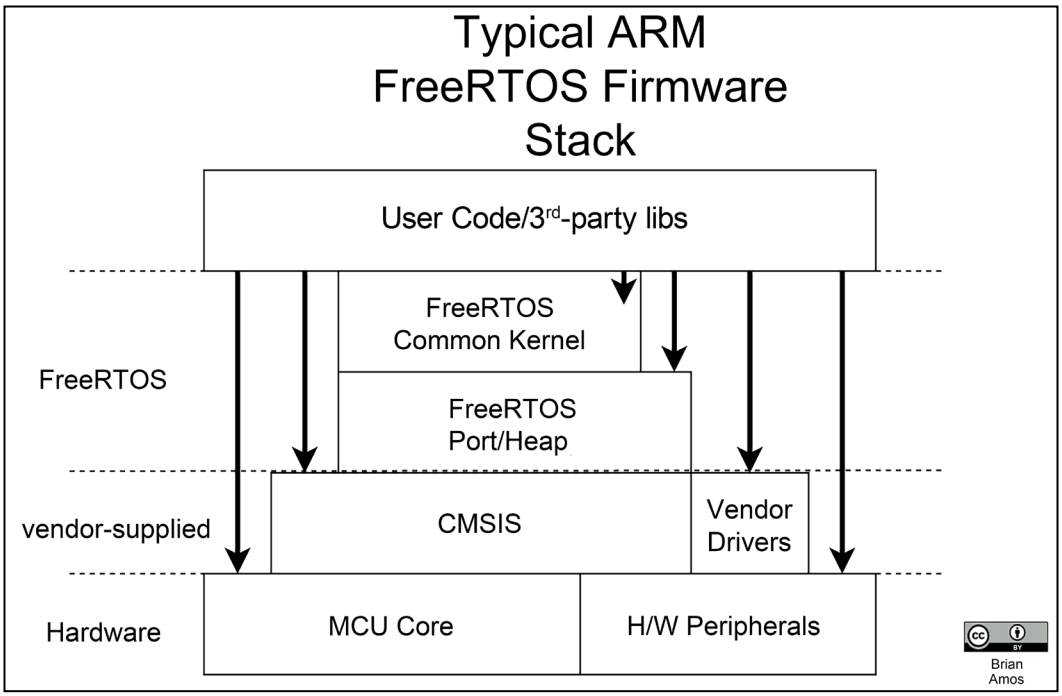


Figure 1.5: Typical ARM FreeRTOS firmware stack

Some noteworthy items are as follows:

- User code is able to access the same FreeRTOS API, regardless of the underlying hardware port implementation
- FreeRTOS does not prevent user code from using vendor-supplied drivers, the CMSIS, or raw hardware registers

Having a **standardized API** that is consistent across hardware means code can be easily migrated between hardware targets without being constantly rewritten. The ability to have code directly control hardware also provides the means to write *extremely* efficient code when necessary (at the expense of portability).

Now that we know what an RTOS is, let's have a closer look at when it is appropriate to use an RTOS, and when it is not. And, when it is appropriate, we'll look at determining the type of RTOS to use.

Choosing an RTOS

Occasionally, when someone first learns of the term *real-time OS*, they mistakenly believe that an RTOS is the only way to achieve real-time behavior in an embedded system. While this is certainly understandable (especially given the name), it couldn't be further from the truth. Sometimes, it is best to think of an RTOS as a *potential* solution, rather than *the* solution to be used for everything. Generally speaking, for an MCU-based RTOS to be the ideal solution for a given problem, it needs to have a Goldilocks-level of complexity—not too simple, but not too complicated.

If there is an *extremely* simple problem, such as monitoring two states and triggering an alert when they are both present, the solution could be a straightforward hardware solution (such as an AND gate). In this case, there may be no reason to complicate things further, since the AND gate solution is going to be very fast, with high determinism and extreme reliability. It will also require very little time for development.

Now, consider a case where there are only one or two tasks to be performed, such as controlling the speed of a motor and watching an encoder to ensure the correct distance is traversed. This could certainly be implemented in discrete analog and digital hardware, but having a configurable distance would add some complexity. Additionally, tuning the control loop coefficients would likely require twiddling the potentiometer settings (possibly for each individual board), which is undesirable in some or most cases, by today's manufacturing standards. So, on the hardware solution side, we're left with a CPLD or an FPGA to implement the motion control algorithm and track the distance traveled. This happens to be a very good fit for either since it is potentially small enough to fit into a CPLD, but in some cases, the cost of an FPGA might be unacceptable. This problem is also handled by MCUs regularly. If existing in-house resources don't have the expertise required with hardware languages or toolchains, then a bare-metal MCU firmware solution is probably a good fit.

Let's say the problem is more complicated, such as a device that controls several different actuators (for example, a motor or a solenoid valve). The device reads data from a range of sensors and stores those values in local storage. Perhaps the device also needs to sit on some sort of network, such as Ethernet, Wi-Fi, a **controller area network (CAN)**, and so on. An RTOS can solve this type of problem quite well. The fact that there are many different tasks that need to be completed, more or less asynchronously with one another, makes it very easy to argue that the additional complexity the RTOS brings will pay off.

Some RTOS tasks are lower-priority and more complex, such as networking and filesystem processing. Other tasks are time-critical, such as controlling actuators and reading sensors. The RTOS helps us to ensure the lower-priority and more complex tasks won't interfere with the time-critical tasks. Also, in many cases, there may be some form of control system that generally benefits from being run at well-defined time intervals—a strength of the RTOS.

Consider a system similar to the previous one, but now there are multiple networking requirements, such as serving a web page, dealing with user authentication in a complex enterprise environment, and pushing files to various shared directories that require different network-based file protocols. This level of complexity *can* be achieved using an RTOS, but again, depending on the available team resources, this might be better left to a full-blown OS to handle (either an RTOS or a general-purpose OS) since many of the complex software stacks required already exist. Sometimes, a multi-core approach might be taken, with one of the cores running an RTOS and the other running a general-purpose OS.

By now, it is probably obvious that there is no definitive way to determine exactly which real-time solution is correct for *all* cases. Each project and team will have its own unique requirements, backgrounds, skill sets, and contexts that set the stage for this decision. There are many factors that go into selecting a solution to a problem; it is important to keep an open mind and choose the solution that is best for your team and project at that point in time.

Summary

In this chapter, we talked a lot about real-time systems. We defined a real-time system and gave some examples of real-time systems in the real world. At this point, you should understand how real-time systems have different timing requirements and how there are different consequences for not meeting those timing requirements.

We then looked at the ways to use hardware and software to implement real-time systems. We saw that some real-time systems can just use hardware to meet their timing requirements. On the other hand, some real-time systems can get away with just using software on a general-purpose OS. In between those extremes is the focus of this book: applications that depend on an RTOS and run on an MCU.

After better understanding real-time systems, we defined an RTOS and described how it differs from a general-purpose OS. We explored the difference between hard, firm, and soft real-time systems. It is important to remember that the timing requirements of the real-time system determine what type of RTOS is necessary to achieve the desired performance.

We discussed the types of RTOSes that are available and why FreeRTOS was chosen for this book. FreeRTOS is mature and is used widely in industry. It is also free! And, learning FreeRTOS is a way to better understand RTOSes in general.

Finally, we looked at when it is appropriate to use an RTOS, and when it is not. And, when appropriate, we looked at determining the type of RTOS to use.

Upcoming chapters show how FreeRTOS works, and how to use it. The chapters are illustrated with example programs that run on an ST development board. The next two chapters present the development board, and also, the development tools (e.g., the IDE) that we will be using to run the example programs.

2

Introducing the Development Board

For any embedded system project, the development system typically consists of the target hardware (in this case, a development board) and the software development tools needed to create the software image that will run on the target.

In this chapter, we will take a closer look at the target hardware used for this book, the STM32 Nucleo-F767ZI development board. The next chapter explores the software development tools that will be used with the development board. The development board will be referred to as the *dev-board* or just the *board*.

The chapter's topics are:

- Dev-board overview
- Understanding the dev-board's components
- Understanding the hardware documentation

Dev-board overview

The dev-board used in this book is the STM32 Nucleo-F767ZI, from STMicroelectronics. (STMicroelectronics also goes by the name "ST," which will be used, for brevity.) The dev-board is shown in *Figure 2.1* (It is ST's picture of the Nucleo-F767ZI. Though, it is actually a NUCLEO-F207ZG, which looks very similar.) This section describes the dev-board's overall architecture and capabilities, as well as the dev-board's components.

In the book, the dev-board is used to run FreeRTOS and the book's example programs. The discussion here is limited to aspects of the board that are addressed in the book. It is assumed the reader has some familiarity with PC architecture and general purpose OS internals.

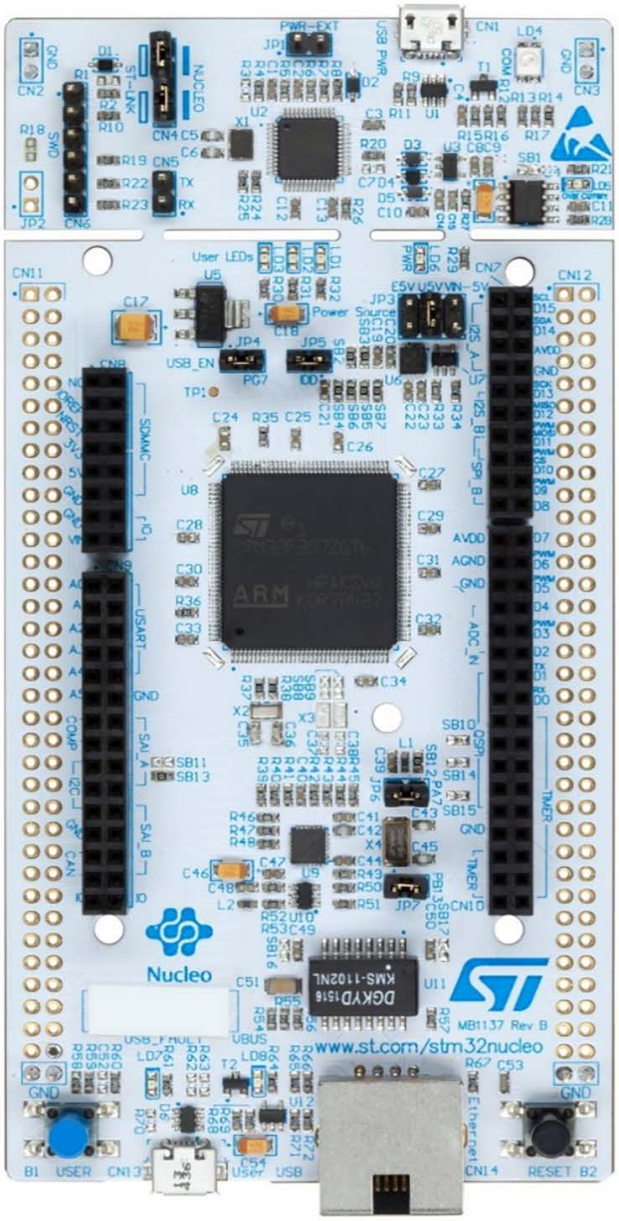


Figure 2.1: The dev-board, an STM32 Nucleo-F767ZI.

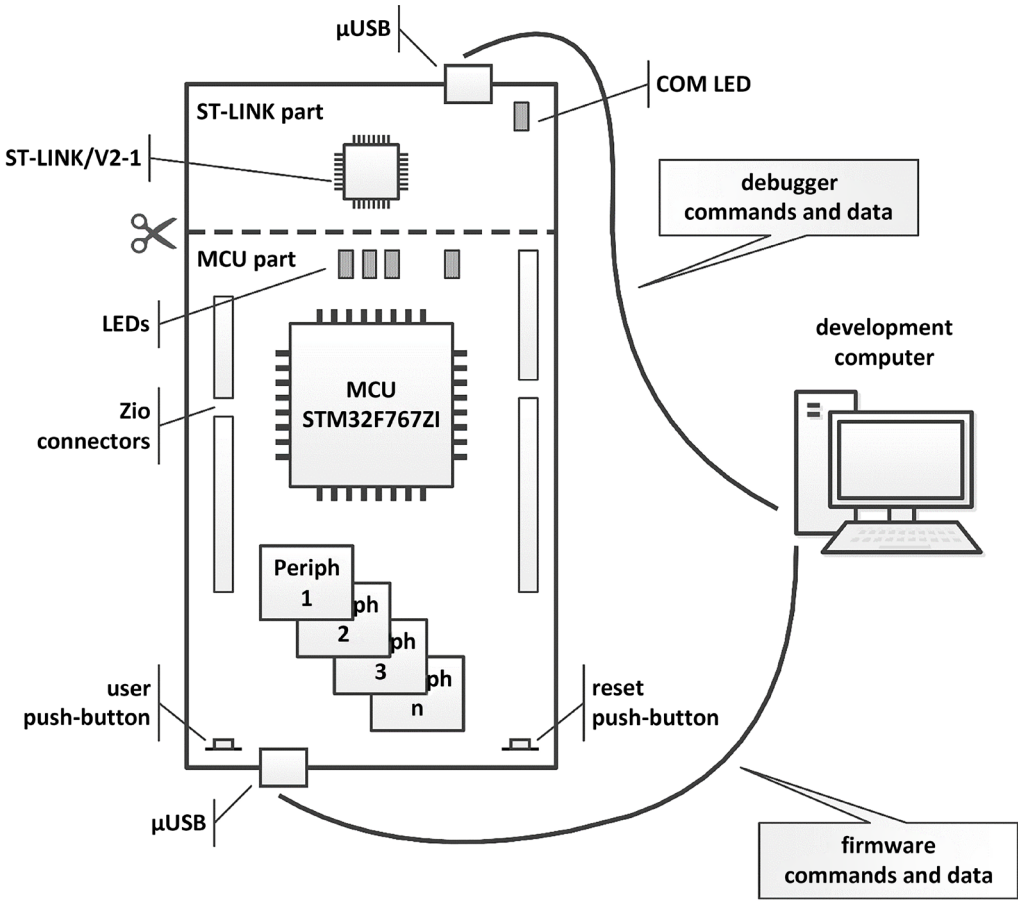


Figure 2.2: The dev-board's architecture and use

The dev-board's primary component is the **microcontroller unit** (**MCU**), and it resides in the MCU part. The MCU is a STM32F767ZI, from ST.

MCU vendors often provide a dev-board for their MCUs, so users can explore the MCUs' capabilities, with the hope that the MCUs will be designed into the users' products. Dev-boards are a quick and easy way for a product-design team to begin software development before the custom target-hardware is ready. These boards also provide a prototyping playground where algorithms may be explored and tested using actual hardware. Occasionally, some dev-boards are integrated into a product, saving the development team the effort of designing the circuits and the PCB surrounding the MCU (this is a little risky, however, as it is possible the vendor could discontinue the development board).

Now that we have introduced the dev-board, let's take a look at its main components.

Understanding the dev-board's components

The dev-board contains the hardware components and interfaces needed to support a wide variety of embedded system applications. The dev-board contains hardware components such as LEDs, push-buttons, USB, Ethernet, power regulation, etc. The dev-board also has hardware interfaces, including pins and cable connectors. One use for the pins is to connect external components to the MCU (e.g., a display, sensors, power, other development-boards, etc.). Another use for the pins is to provide communication links to the MCU, such as serial, USB, or Ethernet. The following sections explore some of the dev-board's main components, starting with the MCU.

The microcontroller unit (MCU)

An MCU is an **integrated circuit** (**IC**), and it is the dev-board's main component. MCUs are often used in embedded systems to provide the primary processing capabilities. Typically, there is much thought and time spent choosing the proper MCU for an application. The capabilities, features, and ecosystem of an MCU significantly impact the overall development effort, the necessary hardware surrounding the MCU, and the ability to implement certain software algorithms.

The ST Nucleo-F767ZI dev-board contains an STM32F767ZI MCU. This MCU can be used in a wide range of applications, including:

- Motor drive and application control
- Medical equipment
- Industrial applications, such as **programmable logic controllers** (**PLCs**), inverters, and circuit breakers
- Printers and scanners
- Alarm systems, video intercom, and **HVAC**
- Home audio appliances
- Mobile applications and the Internet of Things
- Wearable devices, such as smartwatches

Figure 2.3 shows the STM32F767ZI MCU and its primary components (the MCU has additional components which are not shown). As with most MCUs, this MCU has a **central processing unit (CPU)** and system memory, including flash and RAM. The CPU is an ARM Cortex-M7. This MCU contains 2 MB of flash memory and 512 KB of SRAM.

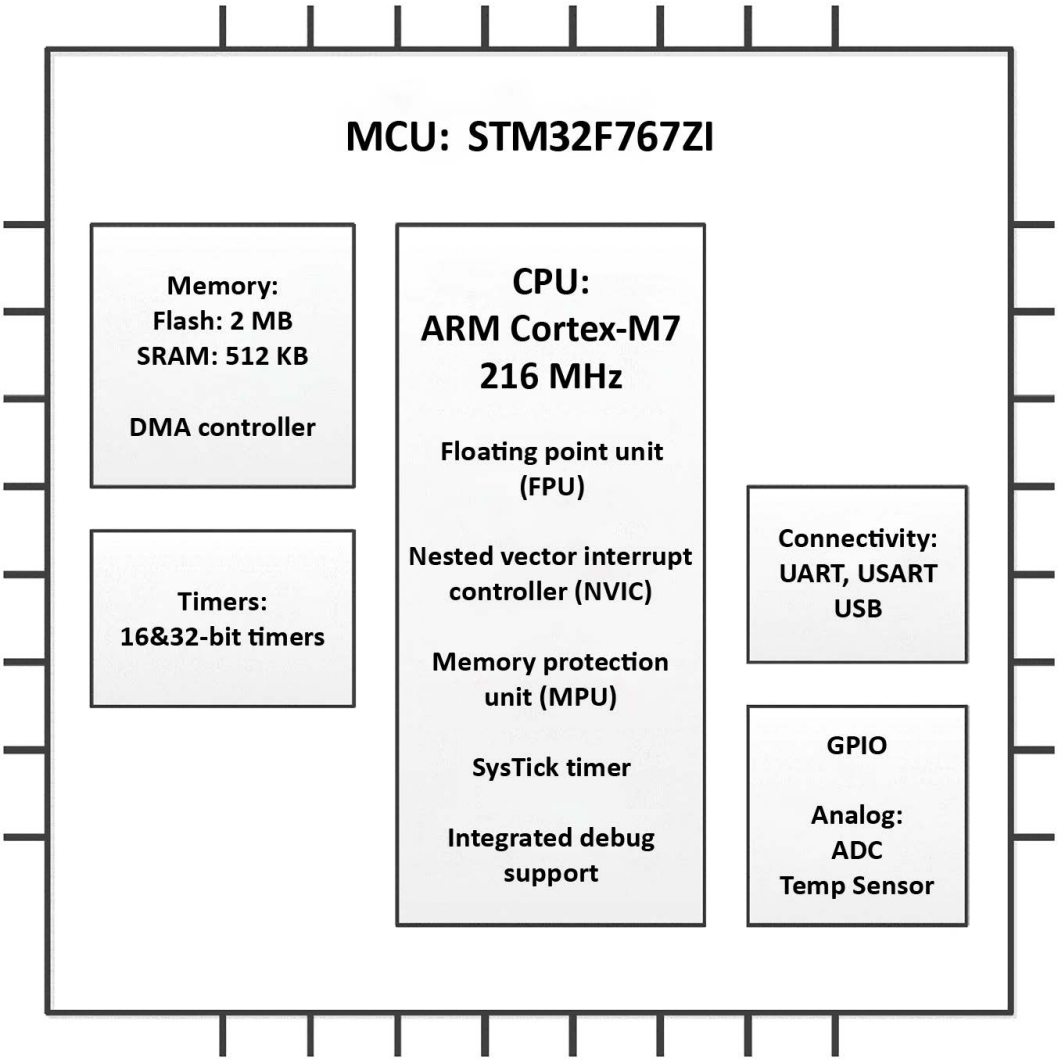


Figure 2.3: The MCU and its primary components

There are other MCUs offered by ST that use the same CPU, but they have reduced flash memory and/or SRAM. A vendor often offers the same CPU with different memory sizes in order to offer users different price points and package sizes. The MCU also has peripherals that are commonly needed in embedded systems, such as **analog-to-digital converters (ADCs)**, and communication devices, like UARTs and USB units. Again, vendors often offer the same CPU with different peripherals, in order to offer users different price points and package sizes.

One reason the MCU is the main component of an embedded system is that it runs the software that provides the desired system capabilities. This software is typically referred to as ***firmware*** because it is stored in the system's flash memory and runs on the system's MCU.

Most of the book's example programs consist of system software and application code. The system software includes FreeRTOS and system libraries, such as CMSIS and the ST **Hardware Abstraction Layer (HAL)**. The application code runs on FreeRTOS. Both the system software and application code are in source-code format. It is compiled and built, and an executable file is created. The executable file's contents are then stored in the MCU's flash memory. The firmware executes on the MCU's CPU, typically from its flash memory (the firmware can also execute from RAM).

Some of the notable peripherals of the STM32F767ZI include:

- **Timers**: 16- and 32-bit timers. One of the book's example programs uses a 16-bit timer in PWM mode to blink an LED.
- **GPIO**: each GPIO pin can be configured by software as input, output, or peripheral alternate function.
- **Direct memory access (DMA)**: provides efficient transfer of data between memory and peripheral registers or between memory and memory, without requiring CPU intervention.
- **Connectivity peripherals**: used in the book's example programs to facilitate communication between the dev-board and the development computer, for example, USART and USB peripherals.

The CPU

The MCU's CPU is an ARM Cortex-M7. The CPU is part of the MCU. The MCU, including the CPU, is made by ST. The ARM company does not make CPUs. Instead, they write specifications for CPUs, and other companies make the CPUs, under a license purchased from ARM. ARM specifies a number of different CPUs for various uses, especially in MCUs.

There are market incentives for MCU makers to use ARM CPUs, such as the ARM Cortex-M7. It creates better economies of scale than if MCU makers used dissimilar CPUs, such as their own proprietary CPUs. For example, if several different MCUs use an ARM Cortex-M7, firmware that runs on one of those MCUs can be more easily ported to run on the others. Also, when different MCUs use the same type of CPU, it reduces costs for producing development tools for those MCUs, such as **integrated development environments** (**IDEs**) and debuggers. Further, when different MCUs use the same type of CPU, it reduces costs for creating external components for use with those MCUs, such as sensors and controllers.

The ARM Cortex-M7 is a 32-bit CPU. On our dev-board, its clock speed is 216 MHz. That may seem slow by current standards, as inexpensive laptops have clock speeds around 2 GHz, which is about 10 times faster. However, laptop CPUs are intended for computing applications that require high processing bandwidth to perform activities, such as graphics manipulation, network communication, etc. Such applications require a lot of computing power. The Cortex-M7, however, is intended for different applications such as signal processing and real-time control. Such applications typically require less processing bandwidth and less computing power. The difference between a laptop's CPU and a Cortex-M7 is like the difference between a muscle car's V8 engine and a motorcycle engine: they both work in a similar way, but it would be overkill to put a V8 engine on a motorcycle!

Figure 2.3 shows the Cortex-M7 CPU with a list of its primary components:

- The **floating point unit** (FPU) greatly speeds-up floating-point arithmetic.
- The **nested vector interrupt controller** (**NVIC**) supports interrupt processing. FreeRTOS interacts with NVIC, in supporting **interrupt service routines** (**ISRs**).
- The **memory protection unit** (**MPU**) is used to ensure that code only accesses the range of memory it is permitted to access. MPUs can help ensure greater system stability and increased security, since an application is less likely to experience unintended consequences from accessing memory it shouldn't. FreeRTOS includes support for MPU-protected tasks.
- The **SysTick timer** is used by FreeRTOS to run its scheduler at fixed intervals, e.g., every millisecond.
- Cortex-M7 implements hardware **debug** features. They provide debugging controls, such as breakpoints, and visibility for the processor and memory.

Other components in the MCU part

The dev-board's MCU part has a few additional components, and they are described here.

The dev-board's user interface is LEDs and push-buttons. Although they are simple, they are very helpful in running the book's programs. There are three user LEDs, and their typical use is to indicate the firmware's execution status, such as when a particular task is running. A system LED indicates the power status. The board has a push-button for user input, and another for system reset. The dev-board's MCU part also has a USB connector. Some of the book's programs require a USB cable from this connector to the development computer. The USB connection provides communication between firmware running on the MCU and an app running on the development computer.

The ST-LINK debug probe

The upper-portion of the dev-board is the "ST-LINK part." In this book, the ST-LINK part is used to:

- Serve as a debug probe for the MCU
- Provide power to the dev-board
- Flash firmware to the MCU

Figure 2.4 illustrates how the "ST-LINK part" works. It is connected to the development computer via a USB cable. The cable provides power to the dev-board. The development computer runs debugging software, which we'll refer to as the **debugger**. The debugger provides standard debugging capabilities for the firmware, while the firmware runs on the MCU. These capabilities include source code debugging, breakpoints, watchpoints (for data access), and trace (the logging of execution activity).

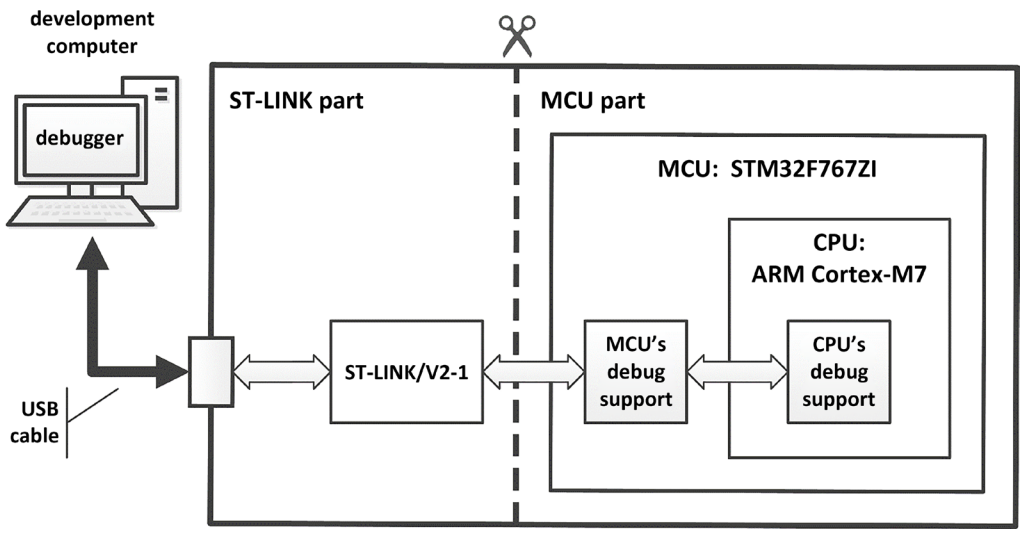


Figure 2.4: The ST-Link part's interfaces

ST provides an IDE called STM32CubeIDE, and it has a debugger that can be used with this dev-board. There are also third-party debuggers that can be used.

For the ST-LINK part, its primary component is the ST-LINK/V2-1, which is a **debug probe** for the MCU. The debugger communicates with the ST-LINK/V2-1, over the USB cable. The ST-LINK/V2-1 communicates with the MCU, over traces on the dev-board.

One use of the ST-LINK/V2-1 is to flash firmware to the MCU. The firmware source code is compiled and built on the development computer, and an executable file is produced. The debugger is used to send the executable file's contents to the ST-LINK/V2-1, which writes the contents to flash memory on the MCU. After the firmware is written to flash memory, there are several ways to run the firmware:

- It can be started via the debugger, which instructs the ST-LINK/V2-1 to run it.
- The dev-board's reset button can be pushed.
- The dev-board can be powered off, then on.

As shown in *Figure 2.4*, both the MCU and CPU have debug support components. The MCU's debugging capabilities are ultimately provided by the Cortex-M7 CPU and its debug support component. The MCU's debug support component provides an interface to the Cortex-M7's debug support component. A debug probe connects to the MCU's debug support component via pins on the MCU. The debug probe accesses the Cortex-M7's debug support component through the MCU's debug support component.

ST provides a standalone debug probe for use with this MCU. This standalone debug probe is also an ST-LINK/V2 debug probe, as shown in *Figure 2.5*. (The picture is from *ST's manual, ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32* (UM1075)) It can be used with an MCU that is mounted on a circuit board in an embedded system. The debug probe's ribbon cable attaches to a cable header on the circuit board, which connects to the proper pins on the MCU. A standard 20-pin JTAG flat ribbon is used, and the JTAG, SWD, and SWV communication protocols are supported. The debug probe's USB cable connects to the development computer. The development computer runs a debugger, which communicates with the debug probe over the USB cable.



Figure 2.5: A standalone implementation of the ST-LINK/V2 debug probe

The dev-board's debug probe is very similar to the standalone debug probe, except that the former is implemented on the dev-board. Instead of a ribbon cable, the dev-board has traces that connect the debug probe to the MCU.

The dev-board's debug probe, the ST-LINK part, can even be separated from the MCU part by cutting the board along the dashed line, as shown in *Figure 2.4*. This way the ST-LINK part can be used as a standalone debug probe, for example, to save space. Of course, wires must be added to connect the ST-LINK part to the MCU part and to power the MCU part.

The debugger can send debugging commands to the ST-LINK/V2-1 debug probe, such as commands to set breakpoints and watchpoints, and commands to get the contents of registers and memory. The debug probe then carries out the command by communicating with the MCU and CPU's debug support components. Debugging information is sent back along this same path, but in reverse, from the Cortex-M7 CPU to the debugger. Examples of debugging information are the values in registers and memory, and an indication that a breakpoint has been reached.

The debug probe communicates with the debug support components in the MCU and CPU. Either the JTAG or SWD communication-protocol can be used. They are high-speed serial protocols that define methods of data exchange and control. In the early history of embedded system development, an **in-circuit emulator (ICE)** was used for debugging. It was an expensive piece of hardware that was used in place of the MCU, in order to emulate the MCU's functions and allow for debugging. Later, a debugging peripheral was used in the MCU, and JTAG was used to communicate with it. SWD is a more recent communication-protocol for ARM CPUs, and one of its advantages over JTAG is that it uses fewer wires (two instead of four).

Another feature of the ST-LINK/V2-1 debug probe is a virtual COM-port interface on USB (VCP). This allows an RS232 serial connection to be created between the firmware on the MCU and an app on the development computer, and the serial connection is emulated over USB. On the development computer, the serial connection appears as a serial port, for example, under Windows Device Manager. This feature is used in some of the book's example programs, as it allows the use of UARTs and serial connections, but without requiring physical serial-adapters and serial cables.

There are different debug-probe capabilities available from vendors. Less expensive models tend to have a slower communication interface between the development computer and the dev-board. Conversely, more expensive models have faster communication interfaces, and they have additional features such as instruction tracing, real time profiling, and real time variable monitoring.

It is important to mention that the ST-LINK/V2-1 chip has its own firmware that is separate from the firmware that is burned into the MCU. ST-LINK/V2-1 has a firmware upgrade mechanism that allows it to update itself through the USB port. Firmware upgrades are made by using STMCube32IDE, which interacts with the ST-LINK/V2-1 firmware upgrade mechanism. Occasionally, a dialog may appear, indicating that there is a firmware upgrade for ST-LINK/V2-1. This firmware update does not impact the firmware on the MCU.

For the book's programs, a third-party debugger will be used, instead of ST's debugger. It is the Ozone debugger, from Segger. Ozone works with Segger's collection of debug probes, called J-LINK. To use Ozone, the firmware on the ST-LINK/V2-1 chip is replaced with Segger's J-LINK firmware. This converts the ST-LINK part of the board into a J-LINK debug probe. *Figure 2.6* provides an illustration.

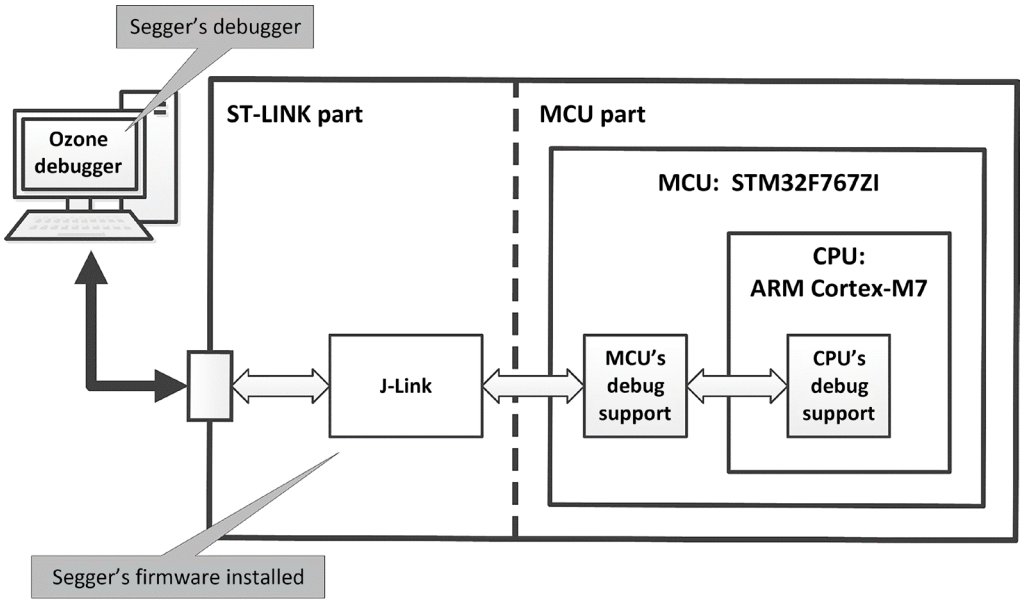


Figure 2.6: Using Segger's Ozone debugger and J-Link debug probe

The J-Link debug probe and Ozone debugger are used throughout the book, to show how the example programs work. And there will be an opportunity to use these tools extensively, with the book's example programs.

In this section, we introduced the dev-board's components. If the dev-board and its components seem complex, that's because they are. One indication of the complexity is their extensive documentation. The next section describes the sizable collection of documents for the dev-board and its components, as well as how to locate the documents and work with them.

Understanding the hardware documentation

When working with a dev-board, it's essential to have the hardware documentation handy. This involves finding and downloading dozens of PDF documents, and also storing and organizing them. It is a formidable task, as the dev-board, MCU, and CPU each have their own documents, including data-sheets, user manuals, and reference manuals.

Appendix B lists the primary documentation related to the dev-board and its components. Over 30 PDF documents are listed, and over 25 web-pages. In reading the book and working with the example programs, it is very helpful to have these PDF documents downloaded and accessible. If Adobe Acrobat Reader is used, storing the documents in a single directory can be useful to search across all documents.

The remainder of this section describes how ST's products and documentation are organized for MCUs and dev-boards. Also described is how the model names encode information about the model's families and features.

MCU families

ST has a huge collection of MCUs and dev-boards, numbering in the hundreds. The product-information and documentation for them are voluminous, and the way they are organized is complex. The MCUs are organized into groups called "families," and a family can be subdivided into additional families.

The MCU we're using is a model STM32F767ZI, and it belongs to five MCU families. For example, it belongs to the *STM32* family of MCUs, which are 32-bit and have an Arm Cortex-M processor. It also belongs to the *STM32F7* family of MCUs, which are "very high-performance" and have an Arm Cortex-M7 processor. The STM32F7 family is a subset of the STM32 family.

The STM32F767ZI MCU has its own web page. The link is: `https://www.st.com/en/evaluation-tools/nucleo-f767zi.html`

The MCU's web-page has product information, and links to primary documents, such as the user-guide and reference manual. Documents for MCUs typically apply to an MCU family, and not just one MCU model. For instance, the reference manual for STM32F767ZI applies to all STM32F76xxx MCUs and all STM32F77xxx MCUs.

For the STM32F767ZI MCU, three of its MCU families have their own web page (the links are in the following section and in *Appendix B*). An MCU family's web page has product information and links to related documents. In our experience, there can be important MCU documents that are not listed on an MCU model's web page but are listed on one of its MCU family web pages. So, when searching for an MCU model's documentation, it is necessary to look at the MCU model's web page, and its families' web pages.

For example, one type of MCU documents is "Application Notes," which are how-to guides for specific applications. For the STM32 family, one of the family's documents is the Application Note "STM32 microcontroller debug toolbox" (AN4989). That document describes debugging tools for STM32 MCUs. Currently, that document is listed on the web page for the STM32 family but not on the web page for the STM32F767ZI MCU.

Another way to find documents for an MCU is to use ST's website and its search interface: `https://www.st.com/content/st_com/en/search.html`

To search, enter the MCU's model name or a family name. In the search results, select the type of publication, for example, **Application Notes**. The figure below is from a search for **stm32f76 timers**, and the publication type "**Application Notes**" was selected.

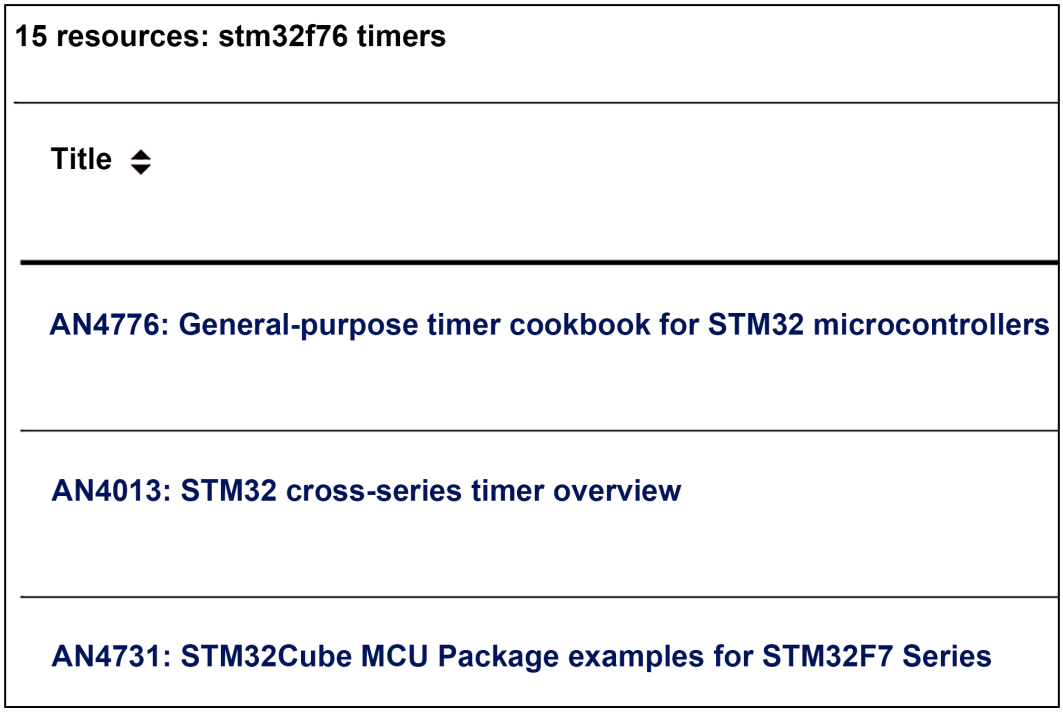


Figure 2.7: Search results from ST's website

MCU model names

An MCU's model name encodes information that specifies the model's families and features. The encoding is shown below for our MCU, an STM32F767ZI. The encoding scheme is specified in the MCU's datasheet, in the section "Ordering information." For the MCU model, its families are specified on the model's web-page (in the "breadcrumb" links) and in the model's datasheet. (For our MCU, both sources had to be consulted.)

This list shows how the **STM32F767ZI** model name specifies the MCU's families and unique features:

- **STM32**: the topmost MCU family. It's a collection of 32-bit MCUs, and they use an Arm Cortex-M processor
    - `https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html`

- **STM32F**: "F" specifies the product type, which is "general purpose."
- **STM32F7**: an MCU family. It's a subset of the STM32 family. This family's MCUs are "very high-performance," and they have an Arm Cortex-M7 processor
    - `https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html`
- **STM32F7x7**: an MCU family. It's a subset of the STM32F7 family. This family's MCUs have a double-precision floating point unit, run at 216 MHz, and have relatively low power consumption
    - `https://www.st.com/en/microcontrollers-microprocessors/stm32f7x7.html`
- **STM32F76x**: an MCU family. "x" can be 5, 7, 8, or 9
    - Specified in the STM32F767xx datasheet
- **STM32F767**: an MCU family. It's a member of the STM32F76x family. This family's MCUs also have the following peripherals: USB OTG FS/HS, a camera interface, Ethernet, and LCD-TFT
    - Specified in the STM32F767xx datasheet
- **STM32F767ZI**: the MCU model. "Z" specifies the MCU has 144 pins. "I" specifies 2,048 Kbytes of flash memory
    - `https://www.st.com/en/microcontrollers-microprocessors/stm32f767zi.html`

Dev-board families and model names

ST dev-boards use the same organization-scheme as MCUs, for specific models, families and documentation. Also, the dev-board's model name encodes information that specifies the model's families and features. For the NUCLEO-F767ZI dev-board, the families are:

- **NUCLEO**: a family of dev-boards. The family's full name is, "STM32 Nucleo Boards." These dev-boards have an STM32 MCU, and a ST-LINK debugger and programmer
    - `https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html`
- **NUCLEO-144**: a sub-family of the NUCLEO family. These dev-boards use a 144-pin MCU
    - Specified in the document "Data Brief: STM32 Nucleo-144 boards" (DB3171).
- **NUCLEO-F767ZI**: the dev-board model. "F767ZI" specifies the MCU is a STM32F767ZI
    - `https://www.st.com/en/evaluation-tools/nucleo-f767zi.html`

In summary, ST's organization-scheme for MCUs and dev-boards is complex. Understanding that organization-scheme makes it easier to find product information and documentation, and to understand the model names. For a particular MCU or dev-board, investigation is needed to determine the families it belongs to. Documentation can then be found at the model's web-page and at its families' web-pages. Documentation can also be found via ST's search interface, by specifying the model name or a family name.

Summary

This chapter introduced the dev-board that we'll use throughout the book. The STM32 Nucleo-F767ZI dev-board is built around the STM32F767ZI MCU. This MCU can be used in a wide range of applications, such as motor control, medical equipment, and printers. The MCU's CPU is a 216 MHz ARM Cortex-M7. It has 2 MB of flash memory, and peripherals such as serial communication units, a USB unit, and ADCs.

The dev-board also has a built-in ST-LINK debug probe. Debugging-software runs on a development computer, and it uses the debug probe to provide debugging capabilities for the MCU. For our use, we will load firmware onto the debug probe to make it work like a SEGGER J-Link.

The next chapter presents the software development tools that we'll use to create the software that runs on the dev-board.

3

Introducing the Development-Tools

This chapter presents the software-development tools that we'll use in the book. These tools are used with the book's example-programs, to run those programs and understand them.

The tools are ST's integrated development environment (STM32CubeIDE), and the SEGGER debugging-tools Ozone and SystemView. In addition, SEGGER's J-Link debugger will be flashed to the dev-board's debug-probe. Detailed instructions are provided here for installing these programs and for using the tools with an example-program. Instructions are also provided for installing and using the book's example-programs.

The software-development tools can be run on Windows, Linux or MacOS. A Windows PC is required to run the programs that flash J-Link to the dev-board's debug-probe, though that is only done once. The OS versions that can currently be used are described in the text-box below. In the future, it's likely that newer versions of these tools, programs and OSes can be used. The book's code-repository has the latest information on that:

`https://github.com/PacktPublishing/Hands-On-RTOS-with-Microcontrollers-Second-Edition`



Three software-development tools are used: **STM32CubeIDE**, **Ozone** and **System-View**.

All of them can be run on: Windows (10, 11), Ubuntu (LTS 22.04, 24.04), Fedora (41), and MacOS (14, 15).

The PC processor that can be used with all of these tools is x64 (not ARM64).

The book shows the software-development tools' installation and use on a Windows PC (with an x64 processor). In creating the book, we did limited testing on Linux. We haven't used the software-development tools on MacOS.

Currently, to run the programs that flash the J-Link debugger, a Windows PC is needed (Windows 10, 11). Again, this is only done once. An x32 or x64 processor is required (ARM64 can't be used).

Appendix A provides a quick-reference for using the software-development tools (i.e., a "cheat-sheet"). It summarizes the tool-use instructions from the present chapter.

It's recommended that the development-tool documentation be downloaded and accessible, e.g., reference manuals and user guides. The documentation is listed in *Appendix B*. There are about half-a-dozen documents for the present chapter.

This chapter has three major sections:

- Using the book's instructions and example-programs
- The IDE and the example-programs
- The debugging tools: J-Link, Ozone and SystemView

Using the book's instructions and example-programs

Like most programming books, this book has a plethora of detailed instructions for writing programs, using development-tools, and working with the example-programs. In reading the book and working with its programs, the reader will inevitably encounter at least some problems. The problems can be due to the reader's mistakes in following the book's instructions, such as typographic errors made in setting-up a tool's configuration file. There will inevitably be some errors in the book and example-programs, given the amount of technical detail. Another source of potential problems is bugs in the development-tools and system-libraries. We have encountered such bugs in developing the book's example-programs.

Diagnosing problems like these can be challenging, especially for those who are new to embedded-systems development. Several resources are available for diagnosis, and they are summarized below.

It's highly recommended that the tool and system documentation be downloaded and accessible, e.g., reference manuals and user guides. They can be helpful for troubleshooting. Some documents have diagnostic information and descriptions of error messages. The documentation is listed in *Appendix B*, and it's organized by book chapter. For the development-tools, the tool's help interface provides a link to documentation that is specifically for the tool's version.

A potential cause of problems is using different versions of the development-tools than were used in writing the book. The versions used are listed below. They were used on Windows 11.

- ST STM32CubeIDE, Version 1.18.1
- SEGGER Ozone, Version 3.38f
- SEGGER SystemView, Version 3.60d

The book has many screenshots of the development-tools. Some screenshots are from an older version of the tool, as shown in *Figure 3.1* (specifies Ozone v3.22a). However, the window shown is otherwise the same for the tool's version that we are currently using.

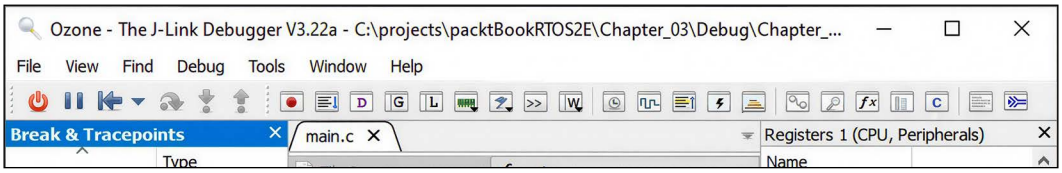


Figure 3.1: Screenshot from an older version of Ozone

The book's website lists corrections for known errors in the book. The URL is provided in the book's preface.

The book's code-repository lists known problems in the code, and it provides fixes. (The code-repository is described later.)

Chapter 16 is on troubleshooting. It includes general troubleshooting techniques, and also, troubleshooting for problems encountered with the development-tools. The code-repository has the latest information on known problems in using the development tools, and workarounds.

The IDE and the example-programs

This section describes the *integrated development environment* (**IDE**) that we'll be using. Instructions are provided for installing and using the IDE, and for installing and using the book's example-programs.

The following topics are presented:

- Introducing STM32CubeIDE
- Installing STM32CubeIDE
- Installing the book's example-programs
- Using the Project Explorer
- Using the editor
- Building a project

Introducing STM32CubeIDE

ST provides an IDE for developing programs that run on its STM32 MCUs and dev-boards, including our NUCLEO-F767ZI dev-board. The IDE is named STM32CubeIDE, and we'll be using it to work with the book's example-programs. The IDE is freely available, and this section describes how to install it, and how to use it to read the book's example-programs and build them.

STM32CubeIDE is based on the Eclipse IDE. It is assumed the reader has some experience with using an IDE, such as Eclipse or Visual Studio. For those unfamiliar with IDEs, the tutorial information here may be sufficient, and additional tutorial information can be found in the *STM32CubeIDE User Guide*. (For the system-documentation cited in the book, *Appendix B* has bibliographic info, including where to find the documents.)

The book uses four features of the IDE: the source-code editor, the build system, the system libraries, and the code generator. The IDE's build-system is based on the GCC toolchain, including GNU's C compiler, C libraries, assembler, and linker. The IDE's system-libraries and code-generator will be described and used in later chapters of this book

One of the IDE's primary features is a debugger, and it is based on GNU's GDB. However, the IDE's debugger isn't used in the book. SEGGER's debugging tools are used instead, and they are described later, in the present chapter.

Installing STM32CubeIDE

For STM32CubeIDE, the version used in the book is 1.18.1. In following along with the book, using a different version may be problematic. If STM32CubeIDE is already installed, it is possible to install multiple versions of it.

STM32CubeIDE can run on Windows, Linux and MacOS, but only x64 OS versions are supported (ARM64 is not currently supported). The OSes that can be used are specified in the ST document, *Release note : STM32CubeIDE release v1.18.1* (RN0114). For all of the books' software-development tools, the OSes that can be used are described at the beginning of the present chapter.

To install STM32CubeIDE, follow these two simple steps:

- Download STM32CubeIDE (`https://www.st.com/en/development-tools/stm32cubeide.html`).
- Install it using the default options.

Now that STM32CubeIDE is installed, we will need to download the book's example-programs and import them into the IDE.

Installing the book's example-programs

To install the book's example-programs we'll look at how they are organized, how to download them, and how to import them into the IDE.

How the book's example-programs are organized

Before showing how to install and work with the book's example-programs, we'll describe how the example-programs are organized within: a) the book, b) the IDE, and c) the book's code-repository.

There are example-programs for the present chapter (chapter 3), and for many of the following chapters.

The IDE's method for organizing programs is based on **workspaces** and **projects**. In the IDE, *a workspace consists of a set of projects*. And, *a project is a directory that contains one or more programs*. A project and its directory are created by using the IDE and its project-creation feature. We'll refer to the project's directory as the **project-directory**. The project-directory's name is also the **project's name.**

For each chapter that has example-programs, the chapter's programs are put in an IDE project. We'll refer to these IDE-projects as **chapter-projects**.

For the chapter-projects, their project-directories are stored in the book's code-repository. The code-repository is shown in *Figure 3.2*, and its URL is: `https://github.com/PacktPublishing/Hands-On-RTOS-with-Microcontrollers-Second-Edition`

For the present chapter, its example-program is in the IDE-project `Chapter_03`. The project-directory is also named "Chapter_03", and the project-directory is stored in the book's code-repository.

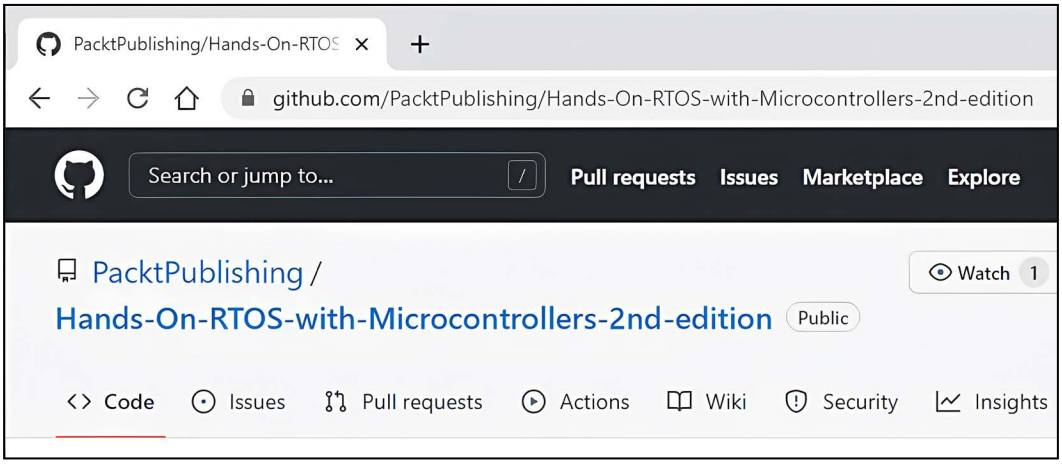


Figure 3.2 : The book's code-repository

Download the example-programs

Create a directory to hold the code-repository's contents. We'll refer to that directory as the **example-programs directory**. The suggested path for the directory is: `C:\projects\packtBookRTOS2E`. That path is used in the book, in referring to the example-programs. The path-name should be short, to avoid exceeding Windows' character limitations for path-names.

Download or clone the book's GitHub code-repository, and put the contents in the example-programs directory. Your example-programs directory should look like *Figure 3.3*.

- Be sure you are using the repository for the second edition of the book (not for the first edition).
- The example-programs directory should contain the repository's top-level contents, e.g., the directories `Chapter_03`, `Chapter_04`, etc. The example-programs directory should not just have a single directory with the repository's name. If you are using `git clone` from the command-line, you can use the option `[<directory>]` (see `https://git-scm.com/docs/git-clone`).
- In Git Bash, these commands will clone the repository's top-level contents to `C:\projects\packtBookRTOS2E`.

```
> cd /c/projects
> git clone "https://github.com/PacktPublishing/Hands-On-RTOS-with-Microcontrollers-Second-Edition.git" packtBookRTOS2E
```

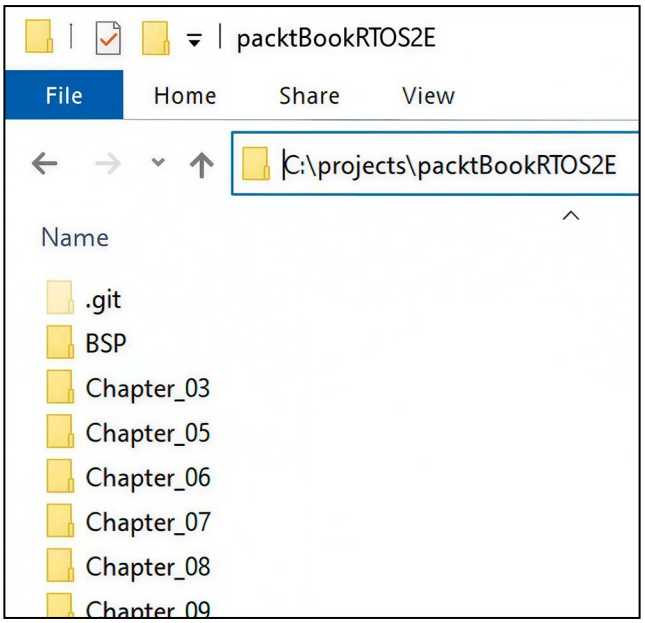


Figure 3.3 : The example-programs directory, with the repository's contents (partial)

Having downloaded the example-programs, the next step is to import them into the IDE.

Import the example-programs into STM32CubeIDE

First, create an IDE-workspace for holding the chapter-projects.

- In the IDE, open the prompt for creating a workspace (*Figure 3.4*). This prompt may appear when the IDE is started, or it can opened by selecting: **File | Switch Workspace | Other...**
- At the **Workspace** prompt, specify the path for the example-programs directory, e.g., `C:\projects\packtBookRTOS2E`.
- Click **Launch**, and the workspace will be created and opened.
- If a different workspace was already open, the IDE will restart. The reason is that only one workspace can be open for each IDE instance.

A workspace has a directory, and we'll refer to it as the **workspace-directory**. In the example just shown, the workspace-directory is `packtBookRTOS2E`, and its path is `C:\projects\packtBookRTOS2E`.

In our case, the workspace-directory is also the example-programs directory.

A **workspace's name** is the workspace-directory's name. In our example, the workspace's name is `packtBookRTOS2E`. However, we'll refer to the workspace using the descriptive name, the **example-programs workspace**.

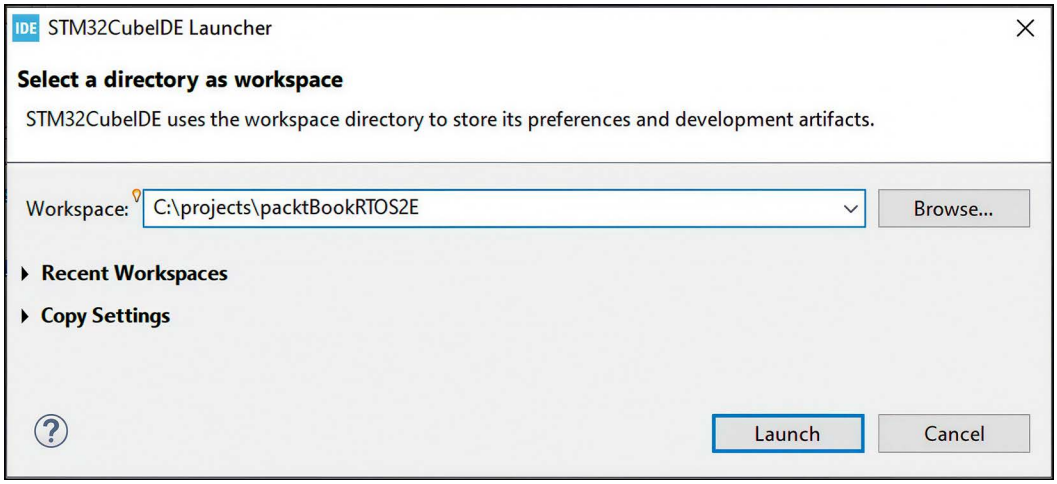


Figure 3.4 : The IDE's prompt to select or create a workspace

Next, import the chapter-projects into the example-programs workspace:

- If the IDE is not already open, then open the IDE and the example-programs workspace. The IDE's prompt to select a workspace is at: **File | Switch Workspace | Other...**
- In the IDE, if the Project Explorer panel is not already open, open it by selecting: **Window | Show View | Project Explorer**.
- If Project Explorer does not open, try clicking on the **Restore** icon, which is just under the **File** menu. Then, open Project Explorer.
- Import all of the chapter-projects:
    - Select: **File | Import**.
    - Select: **General | Existing Projects Into Workspace | Next**.
    - The **Import Projects** prompt will open (*Figure 3.10*).
    - In the text-box **Select root directory:**, specify the path of the example-programs directory.
    - Under **Projects**, the chapter-projects will be listed. Click on **Select All**.
    - The prompt should now look similar to *Figure 3.5*.

- Click **Finish**. (The **Next** button is always gray.)

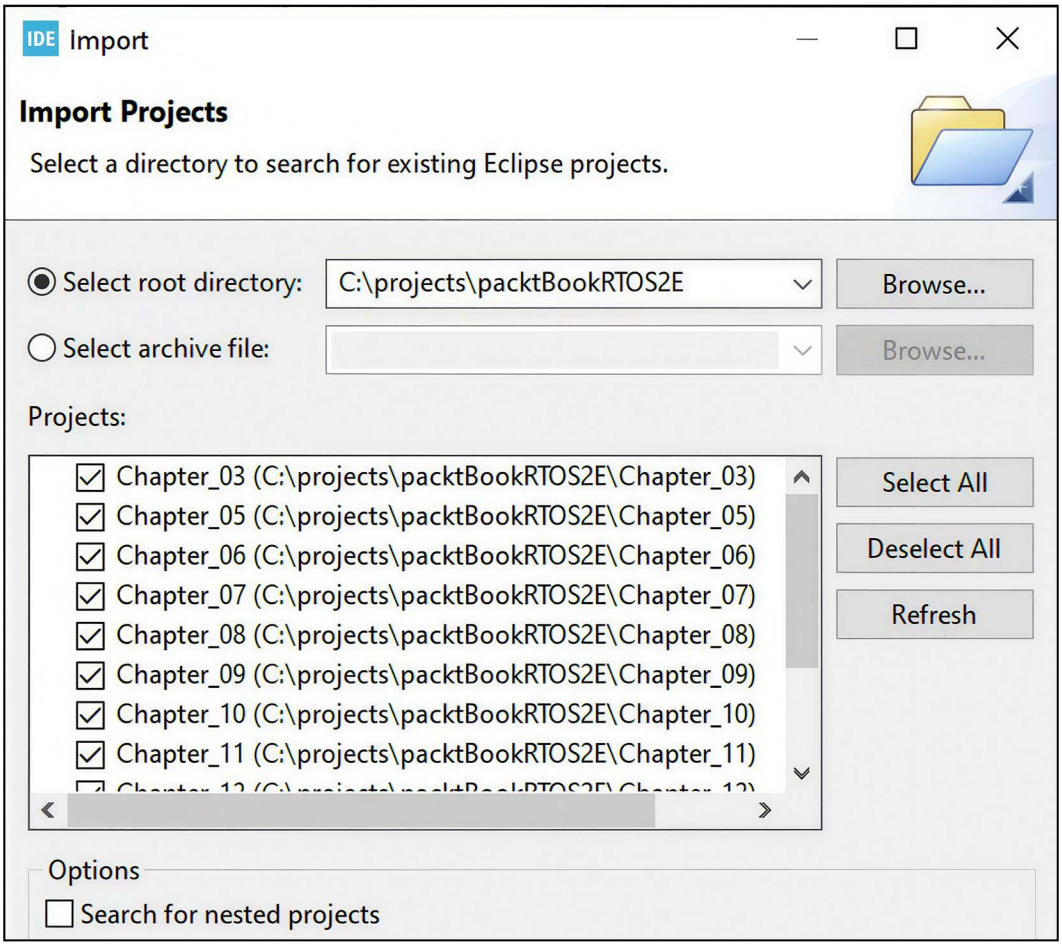


Figure 3.5 : Importing the chapter-projects

The Project Explorer panel should now show that the chapter-projects were imported, as illustrated in *Figure 3.6*. (In Project Explorer, use **Refresh** if needed.) The chapter-projects' names have the form `Chapter_xx`. The project `FreeRTOS_project` should also have been imported (not shown in the figures).

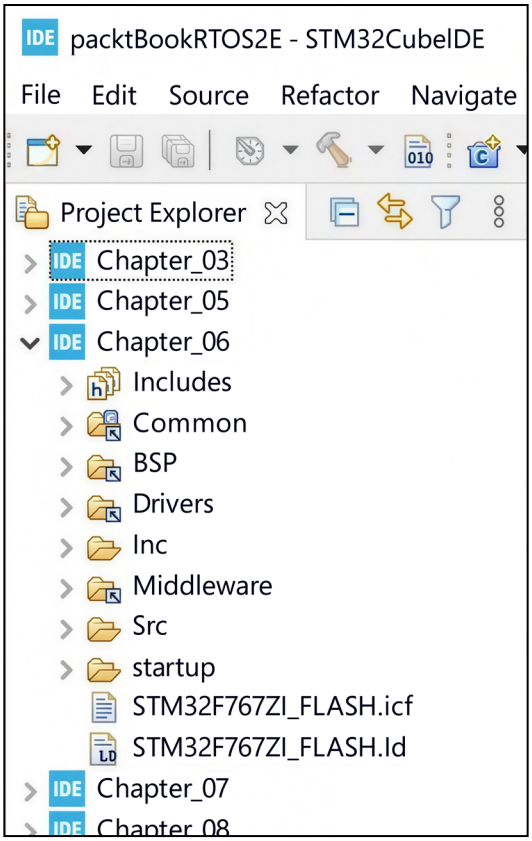


Figure 3.6 : Project Explorer, with the imported chapter-projects

Using the Project Explorer

In the IDE, the active workspace's projects are displayed by the Project Explorer (*Figure 3.6*). The display is hierarchical, and the top-level only shows projects. The top-level does not show other files and directories. For example, the workspace-directory has a top-level directory BSP (*Figure 3.3*), and it's not shown in the top-level of the Project Explorer.

In Project Explorer, a project can be expanded to show the project's files and directories, such as `Chapter_06` in *Figure 3.6*. The project-directory's files and directories are shown, for example, `Inc` and `Src`. Additional files and directories may also be shown, which are not in the project-directory, such as files and directories specified in the project's build-configuration. The directories with an arrow-icon are not in the project-directory. For example, `Common` and `BSP` are specified in the project's build-configuration, and they reside in the workspace-directory.

The IDE saves configuration-data for the workspace and for each project. The workspace configuration-data is stored in the workspace-directory, in the sub-directory `.metadata`. In each project-directory, the project configuration-data is stored in files, such as `.cproject` and `.project`, and in the directory `.settings`. These IDE configuration-directories and configuration-files are not shown in Project Explorer, for all projects.

Using the editor

As an example of using the IDE's editor, we'll open the file `main.c` in the project `Chapter_03`:

- If the IDE is not already open, then open the IDE and the example-programs' workspace, e.g., `packtBookRTOS2E`.
- In the Project Explorer, expand `Chapter_03/Core/Src`. Then, double-click on `main.c`, to open it in the IDE's editor.

`Chapter_03` is a program that flashes the green LED on the dev-board. Scroll down in `main.c`, and you'll see the while-loop that flashes the LED (*Figure 3.7*).

main.c

```
114
115   /* Infinite loop */
116   /* USER CODE BEGIN WHILE */
117   while (1)
118   {
119
120     // The loop-counter is just used for demonstrating SystemView's capabilities
121     loopCounter++;
122     // Write a message to the SystemView app
123     SEGGER_SYSVIEW_PrintfHost("Starting loop-iteration: %u\n", loopCounter);
124
125     // Toggle the green LED
126     HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0);
127     // Wait for 1 second (1000 ms)
128     HAL_Delay(1000);
129
130     // Write a message to the SystemView app
131     SEGGER_SYSVIEW_PrintfHost("Ending loop-iteration: %d\n", loopCounter);
132
133     /* USER CODE END WHILE */
134
135     /* USER CODE BEGIN 3 */
136   }
```

Figure 3.7 : The Chapter_03 project, and its main.c

It's assumed that readers are familiar with IDE editors. A noteworthy feature of this editor is the ability to easily open API source-code. Just right-click on an API call, and select **Open Declaration**.

Building a project

To ensure everything is installed correctly, we'll build the `Chapter_03` project.

- In Project Explorer, select `Chapter_03`.
- Then select: **Project | Build Project**. (Alternatively, in the Project Explorer, right-click on the project and select **Build Project**.)

The build output is shown in the **CDT Build Console**, and there should be no errors or warnings (*Figure 3.8*).

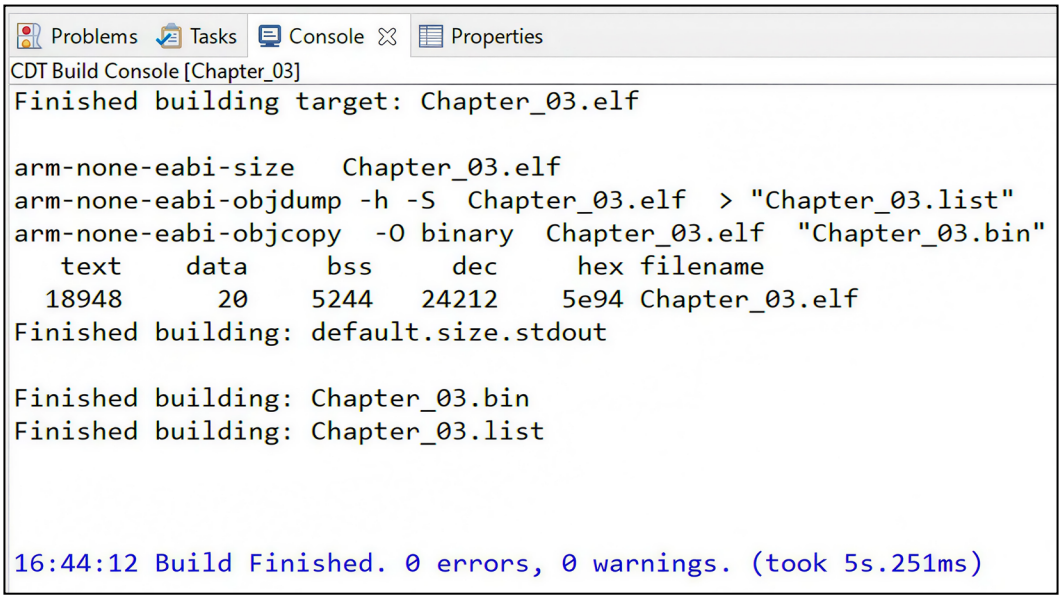


Figure 3.8 : Building the Chapter_03 project

The IDE-project `Chapter_03` only has one build configuration. For projects with multiple build-configurations, the build-project function will build the *active build-configuration*. To set the active build-configuration, select: **Project | Build Configurations | Set Active**. (Alternatively, in the Project Explorer, right-click on the project and make those selections.) *Figure 3.9* shows the two build-configurations for the IDE-project `Chapter_07`.

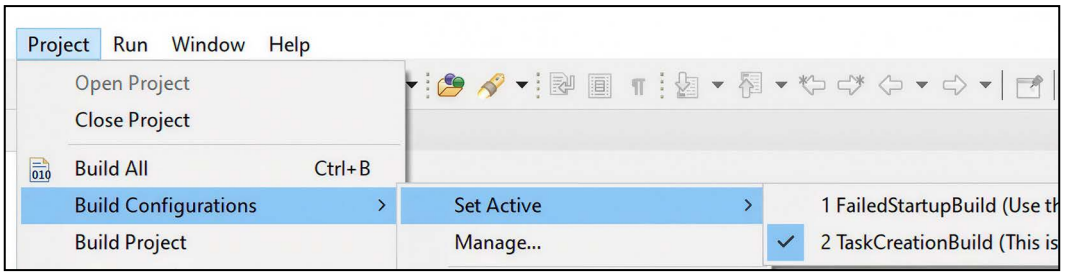


Figure 3.9 : Setting the active build-configuration

Now that the IDE is installed, and the book's example-programs are installed, the next step is to install the debugging tools, and configure them.

The debugging tools: J-Link, Ozone and SystemView

The debugging tools allow the programmer to investigate the execution of the binary image on the target hardware. Debugging tools are essential to real-time software development. So, there is benefit to understanding them well.

In the prior chapter, the dev-board's debug-probe was described, including how it is used with debugging software on the development-computer. The present section describes the debugging software that is used in the book. Also, instructions are provided for installing and using that software.

First, J-Link is presented. It is used to convert the dev-board's ST debug-probe into a SEGGER debug-probe. Next, SEGGER's debugging-tools Ozone and SystemView are presented. They are used throughout the book.

Using SEGGER J-Link

As described earlier, the dev-board has an ST debug-probe called **ST-LINK**. We'll install SEGGER's J-Link firmware on the ST-LINK debug-probe. This will make the ST debug-probe act like a SEGGER J-Link debug-probe. Having a SEGGER J-Link debug-probe allows us to use SEGGER's debugging-tools **Ozone** and **SystemView**.

The whole process of installing the SEGGER J-Link debug-probe is only done once. A Windows PC is required (Windows 10 or 11), with an x32 or x64 processor (not ARM64).

For the J-Link that is installed on an ST-LINK debug-probe, it appears that SEGGER provides very limited documentation. The applicable J-Link documentation is listed in *Appendix B*. We were not able to find a datasheet for this J-Link model.

The next section describes how to download and install the needed utilities. The subsequent section describes how to convert the ST-LINK to a J-Link.

Download and install the utilities

Detailed instructions for installing J-Link are available from SEGGER at: `https://www.segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board/`.

Follow those instructions to install J-Link. Notes about the installation are included below, to clarify the process.

1. First, the ST-Link USB driver must be installed. If STM32CubeIDE is installed on the Windows PC, this step can be skipped. Otherwise, download and install the ST-Link USB drivers. The version used for the book is STSW-LINK009 v2.0.2. `http://www.st.com/en/development-tools/stsw-link009.html`.
2. Download and install SEGGER's J-Link utilities for your Windows PC (the "J-Link Software and Documentation Pack"). The version used for the book is v8.30. The default options are fine. `https://www.segger.com/downloads/jlink`.
3. Download SEGGER's ST-Link Reflash Utility. The version used for the book is dated 2019-08-12. It is a 32-bit Windows program. `https://www.segger.com/downloads/jlink#STLink_Reflash`.
4. Unzip the contents of `STLinkReflash_<version>.zip`— it will contain two files:
    - `JLinkARM.dll`
    - `STLinkReflash.exe`

Now, we will convert the ST-Link to J-Link.

Converting ST-Link to J-Link

Follow these steps to upload J-Link firmware to the ST-Link debug-probe on the Nucleo dev-board:

1. Plug in a micro USB cable to `CN1` on the Nucleo board and attach it to your Windows PC.
2. Open `STLinkReflash.exe`.
3. Read through and accept the two license agreements.
4. Select the first option: **Upgrade to J-Link**.

The debugging hardware on the Nucleo board is now effectively a SEGGER J-Link!

Now that a J-Link is present, we will be able to use other SEGGER software tools, such as Ozone and SystemView, to debug and visualize our applications.

Using SEGGER Ozone

SEGGER Ozone is a piece of software that is meant to debug an already-written application. Its capabilities include source-code debugging and support for debugging programs running under FreeRTOS.

Ozone is independent of the underlying programming environment used to create the application. For example, it can be used with programs created via various IDEs, including STM32CubeIDE. Ozone can be used in many different modes, but we'll focus on importing an executable file in *.elf format and cross-referencing the file with source code. This can provide debugging capabilities for a project created with various toolchains, such as C, C++, and other languages.

The following topics will be presented:

- The file-types used
- Installing Ozone
- Configuring the Ozone project-file (.jdebug)
- Using Ozone to download and run the program
- Using Ozone's source-code debugger

Next, let's take a quick look at the file types we will be working with in Ozone.

The file-types used

There are several file-types used when programming and debugging embedded systems. These files are common across many different processors and software products, and they are not exclusive to Cortex-M MCUs or the software used in this book.

Executable and Linkable Format (ELF) files are an executable format that can store more than is possible with the straight *.bin or *.hex files that are commonly flashed to an MCU's ROM. The *.elf files are similar to the *.hex files in that they contain all of the binary machine-code necessary to load a fully functional project onto a target MCU. The *.elf files also contain links to the original source-code filenames and line numbers. Software such as Ozone uses these links to display source-code while debugging the application.

Some of the relevant file-types are:

- ***.bin**: a straight binary file (just 1s and 0s). This file format can be directly "burned" into an MCU's internal flash memory, starting at a given address.
- ***.hex**: usually a variant of the Motorola S-record format. This ASCII-based file format contains both absolute memory addresses and their contents.
- ***.elf**: contains both the executable code as well as a header that is used to cross-reference each memory segment to a source file. This means a single ELF file contains enough information to program the target MCU and also cross-reference all of the source-code used to create the binary memory segments.

  The ELF file does not contain the actual source-code used. It only contains absolute file paths that cross-reference memory segments to the original source-code. This is what allows us to open a *.elf file in Ozone and step through the C source code while debugging.

- ***.svd**: contains information that maps registers and descriptions to the memory-map of the target device. By providing an accurate *.svd file, Ozone will be able to display peripheral views that are very helpful when troubleshooting MCU peripheral code.

  An *.svd file is a file that is usually included with an IDE that supports your MCU. For example, with STM32CubeIDE, a typical location for the needed *.svd file is `C:\ST\STM32CubeIDE_1.5.1\STM32CubeIDE\plugins\ ... \STM32F767.svd`.

For the book's programs, when a program is built, the `*.elf` file is placed within the project-directory. A sub-directory is created with the name of the build-configuration, and the `*.elf` file is placed there. For example, in the `Chapter_03` project, the build-configuration is `Debug`. So, the `*.elf` file is placed in `Chapter_03/Debug`, as shown in *Figure 3.10*:

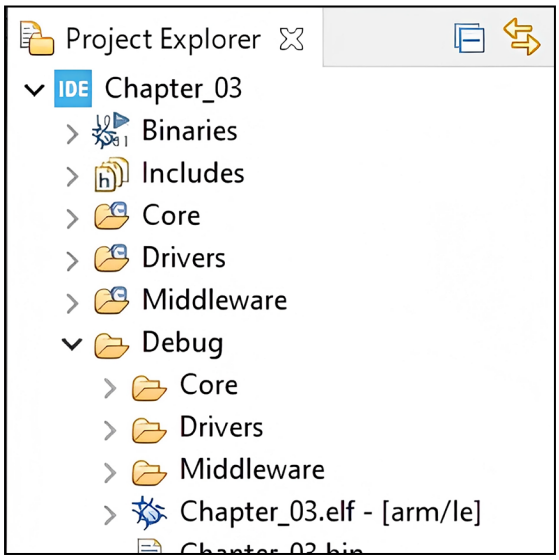


*Figure 3.10 : Chapter_03's *.elf file*

There are other file-types used in embedded-systems development. This is by no means an exhaustive list. It includes the file-types we'll be using with the example-programs.

Installing Ozone

For Ozone, the version used in the book is 3.38f. In following along with the book, using a different version may be problematic. If Ozone is already installed, it is possible to install multiple versions of it.

Ozone can run on Windows, Linux and MacOS. For all of the books' software-development tools, the OSes that can be used are described at the beginning of the present chapter.

To install SEGGER Ozone, follow these two simple steps:

- Download SEGGER Ozone: `https://www.segger.com/downloads/jlink#Ozone`
- Install it using the default options.

Now, let's cover the steps to configure an Ozone project-file and run Ozone to debug a particular program.

Configuring the Ozone project-file (.jdebug)

When Ozone is started, the user is prompted for an **Ozone project-file**, for the program to be debugged. The Ozone project-file specifies the executable-file's path, and the settings for configuring the Ozone debugger and the J-Link debug-probe. Those settings are for the hardware being used, including the MCU and debug-probe.

Ozone project-files have the suffix `*.jdebug`. A template project-file is provided in the example-programs directory, in the sub-directory `templates`, e.g., `C:\projects\packtBookRTOS2E\templates\Ozone-project-file.jdebug`.

Using a standard file-copy, copy that template to the `Chapter_03` project-directory, e.g., to `C:\projects\packtBookRTOS2E\Chapter_03\Ozone-project-file.jdebug`.

To use the Ozone project-file, we have to edit it, to specify the paths for three files: two *.svd files, and an *.elf file.

First, we'll configure the IDE so it can be used to edit *.jdebug files. In the IDE:

- Select: **Window** | **Preferences** | **General** | **Editors** | **File Associations**
- Under **File types**, click **Add...** and enter "*.jdebug".
- Under **Associated Editors**, click **Add...**, and select **Text Editor**

Now we can open `Chapter_03`'s `*.jdebug` file in the editor:

- In the IDE's Project Explorer, expand `Chapter_03`.
- If the file `Ozone-project-file.jdebug` is not shown, run **Refresh**.
- Double-click on the file `Ozone-project-file.jdebug`. This should open the file in the IDE's editor.

Edit the file `Ozone-project-file.jdebug`, to specify the three file-paths, as described in the file's code-comments. Full file-paths are needed for:

- `STM32F767.svd`
- `Cortex-M7F.svd`
- The *.elf file to be run and debugged: `Chapter_03.elf`

For the book's other example-programs, an Ozone project-file will be needed. The *Chapter_03* `*.jdebug` file can be copied. Then, the only edit needed is to specify the path for the *.elf file. Also, the line for FreeRTOS should be uncommented if FreeRTOS is used in the example-program. (FreeRTOS is not used in the *Chapter_03* example-program.)

Some example-programs have more than one build-configuration, as described earlier. An *.elf file is created for each build-configuration. When a build-configuration's *.elf file is to be run, the *.jdebug file must be edited, to specify the *.elf file's path.

Using Ozone to download and run the program

It's time to roll up our sleeves and get our hands dirty! These sections will make more sense if you've got some hardware up and running, so you can follow along and do some exploring. First, let's get set-up to use Ozone:

1. Open the **STM32CubeIDE**, and the example-programs' workspace.
2. In Project Explorer, open the `Chapter_03` project and build it.
3. Configure the Ozone project-file, `Ozone-project-file.jdebug`. All of these steps were described earlier in the present chapter, and they are summarized in *Appendix A*.

Next, we'll open Ozone, and use it to download and run the program:

1. Open Ozone
2. In the pop-up window, select **Open Existing Project**, and specify the path for `Ozone-project-file.jdebug`.
3. Use Ozone to download the `Chapter_03` *.elf file to the MCU, as shown in *Figure 3.11*:

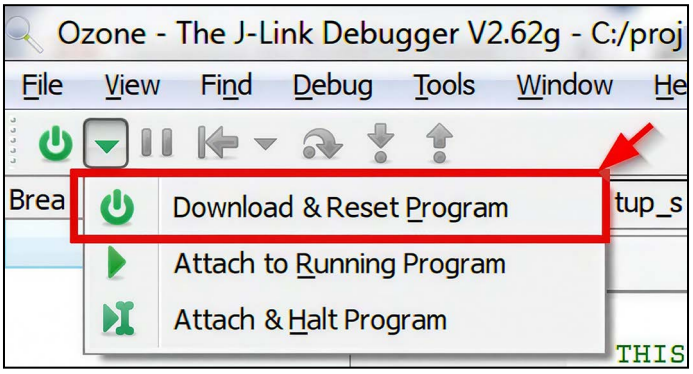


*Figure 3.11 : Downloading the Chapter_03 *.elf file*

4. Click on the play button to run the program. You should see the green-LED flash. The LED is at the top-middle of the dev-board's MCU-part. Ozone should look something like *Figure 3.12*.

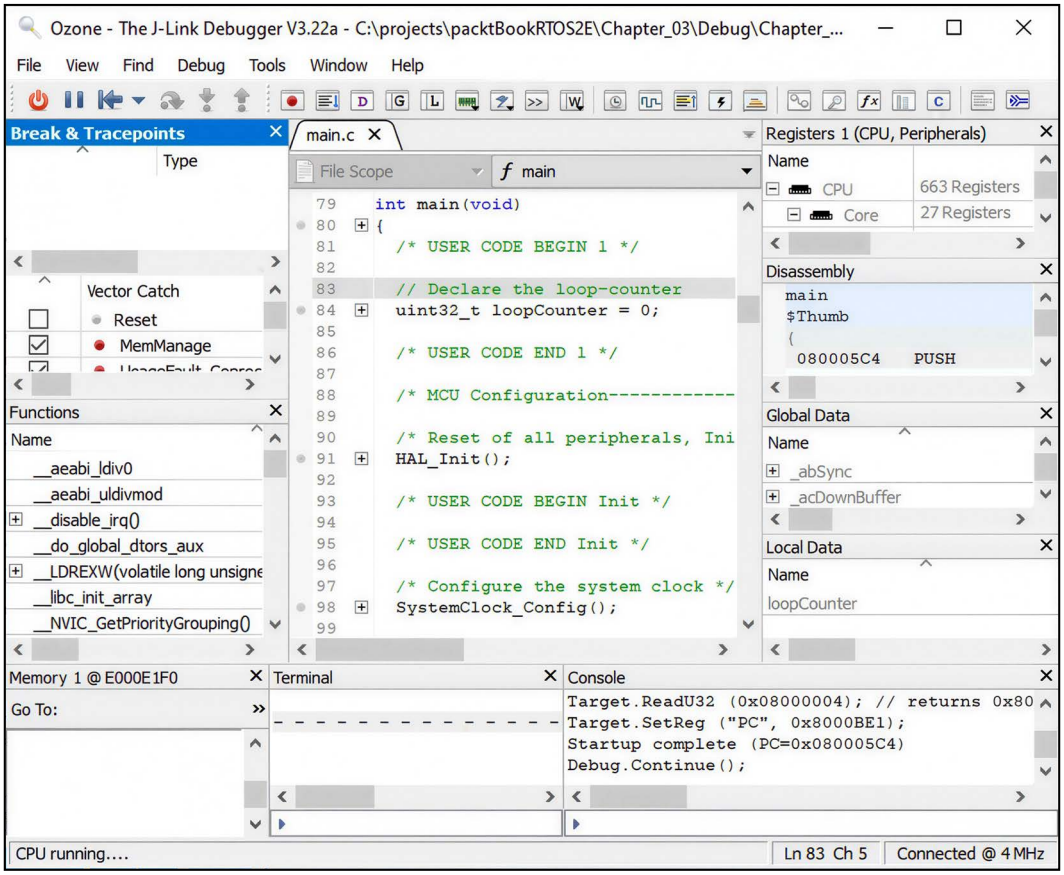


Figure 3.12 : Ozone, running Chapter_03.elf

Those same steps for running Ozone can be repeated for any of the projects included in this book. Some projects have multiple build-configurations, and for them, the intended build-configuration must be made active. The steps for running Ozone are summarized in *Appendix A*.

Using Ozone's source-code debugger

This section highlights some of the primary features in the Ozone source-code debugger. It's assumed that the `Chapter_03` program is running under Ozone, as in the prior section. Ozone should look much like *Figure 3.12*. If `main.c` is not open in the middle panel, it can be opened with: **File | Open**. If any of the other panels are not present, they can be opened using **View**.

First, we'll set a breakpoint in `main.c`, in the function `main()`, at the top of the while-loop.

1. Expand the instruction `loopCounter++;`, to show its assembly instructions. (See *Figure 3.13*.)

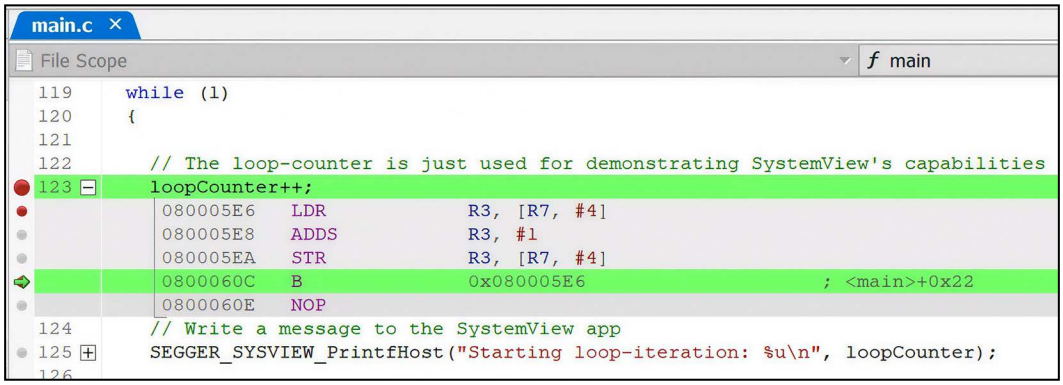


Figure 3.13 : Breakpoint set in main.c, in the function main()

2. Right-click on `loopCounter++;`, and click on **Set Breakpoint**.
3. Breakpoints will be set for two assembly instructions.
4. Within two seconds, the program should pause at one of the assembly-instruction breakpoints. The instruction will be colored green.
5. In the panel **Local Data**, the local-variable `loopCounter` is shown, for the function `main()`. The variable records the number of iterations of the while-loop. (See *Figure 3.12*.)

Press the play button, and the loop-iteration will complete, and the breakpoint will be reached again.

Next, we can step through the loop-body. Experiment with using the debug commands "Step Over" (F10), "Step Into" (F11), and "Step Out" (shift+F11). These commands are available via their hot-keys and also through the menu-option **Debug**. While stepping through the code, notice how the `loopCounter` variable changes, as displayed in the panel **Local Data**. Also, notice how the panel **Call Stack** changes as functions are entered and exited.

In the while-loop body, there are two calls to `SEGGER_SYSVIEW_PrintfHost()`. This API works like `printf()`, but it sends the resultant string to the SystemView app, which runs on the development-computer. The SystemView app displays the string, along with other debugging information. The output from `SEGGER_SYSVIEW_PrintfHost()` is not displayed in Ozone.

`SEGGER_SYSVIEW_PrintfHost()`supports a limited set of `printf()` string-formatting features. For example it supports `%d` for formatting integers, but it does not support `%s` for formatting strings. More information on `SEGGER_SYSVIEW_PrintfHost()` can be found in chapters 7 and 16.

The next section describes how to use SystemView, and how the strings sent by `SEGGER_SYSVIEW_PrintfHost()` are displayed.

Using SEGGER SystemView

SEGGER SystemView is another software tool that can be used with SEGGER debug-probes. There is a SystemView app that runs on the development computer, as do STM32CubeIDE and Ozone. Unlike Ozone, the SystemView app doesn't provide source-code debugging capabilities. Instead, the SystemView app is only a **viewer** and **recorder** for system activity. While the target program runs, system-status data is sent to the SystemView app to be displayed in real-time. The SystemView app also records the data, so it can be displayed later.

In the book, SystemView is used in two ways: 1) to display messages generated in the firmware by `SEGGER_SYSVIEW_PrintfHost()`, and 2) when FreeRTOS is used, SystemView provides a means of visualizing the flow of tasks and interrupts in the system. The present section just shows SystemView use with `SEGGER_SYSVIEW_PrintfHost()`. SystemView use with FreeRTOS is shown in a later chapter.

To use SystemView in a firmware program, SystemView-API calls must be added to the program. The book's example-programs have these API calls. A later chapter shows how to add SystemView API calls when creating a new program.

The next two sections describe how to install and run the SystemView app on a development computer.

Installing the SystemView app

For SystemView, the version used in the book is 3.60d. In following along with the book, using a different version may be problematic. If SystemView is already installed, it is possible to install multiple versions of it.

SystemView can run on Windows, Linux and MacOS. For all of the books' software-development tools, the OSes that can be used are described at the beginning of the present chapter.

To install the SystemView app, follow these steps:

- Download the SystemView app for your OS (`https://www.segger.com/downloads/systemview/`).
- Install using the default options.

Configure the SystemView app to communicate with the debug-probe:

- Open the SystemView app
- Open the app's window named **Recorder Configuration**, by clicking on: **Target | Recorder Configuration | J-Link**.
- Set-up the **Recorder Configuration** as shown in *Figure 3.14*.
- The settings shown are: **J-Link Connection**:USB, **Target Connection**:STM32F767ZI, **Target Interface**:SWD, **Interface Speed**:4000, **Number of cores**:0, **RTT Control Block Detection**:Auto

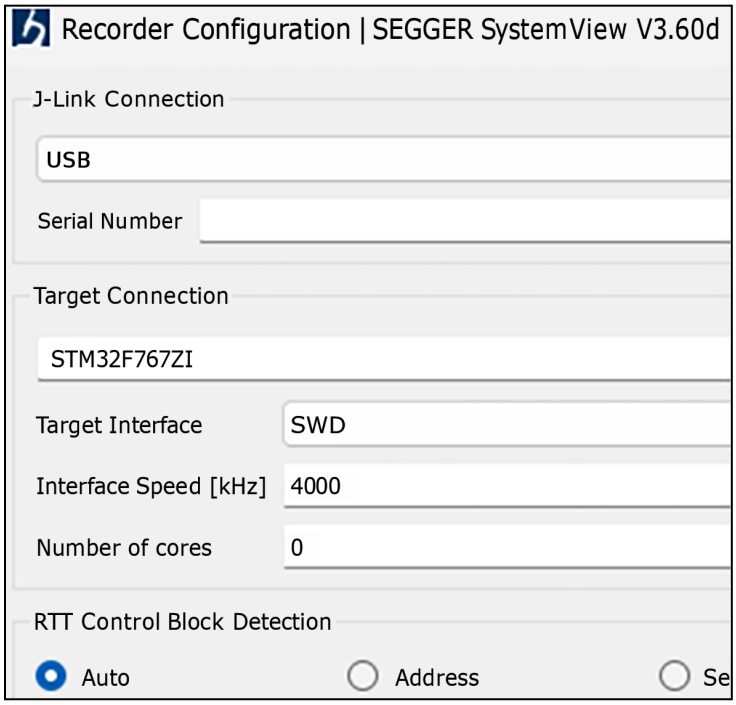


Figure 3.14 : SystemView's Recorder Configuration

In the **Recorder Configuration**, the drop-down box **Target Interface** specifies the MCU debugging-interface to use. For our J-Link debug-probe, SWD is the only interface supported. The highest interface-speed supported is 4MHz, and it is specified in **Interface Speed [kHz]**. We were not able to find a datasheet for this J-Link debug-probe. These specifications were discovered by using SEGGER's J-Link Commander. It is a CLI tool that connects to the debug-probe, and it can be used to configure the debug-probe, and to inquire about the debug-probe's features.

Also in the **Recorder Configuration**, the field **Number of cores** is set to 0. This specifies that all cores are to be found automatically by SystemView.

Running the SystemView app

To use the SystemView app to display data from a target program:

- Ensure the dev-board's debug-probe is connected to the development-computer, via a USB cable.
- Open the app on the development computer.
- If you've not already done so, set-up the **Recorder Configuration** as shown in the prior section.
- Ensure the target program is running on the dev-board, e.g., via Ozone.
- Stop the Ozone debug-session, if it is running. This can be done in Ozone by clicking on the red on/off icon, or by clicking on **Debug | Stop Debug Session**. Stopping the Ozone debug-session is not required here. In general, it is done to avoid overflow with SystemView Recorder, e.g., when running FreeRTOS. (SystemView overflow is described later, in chapter 6.)
- In the app, start the SystemView Recorder by clicking on the **Play** button (**Start Recording**). System-data from the target program will be displayed as it arrives, and also, the data will be recorded.
- To see the messages generated by `SEGGER_SYSVIEW_PrintfHost()`, click on the **Terminal** tab.
- To make the timestamp format consistent among the windows, click on: **View | Display recording time**.

The SystemView app should now look somewhat like *Figure 3.15*. The top-left pane is the **Event List**. The top-right pane is the **Terminal** window. Messages generated by `SEGGER_SYSVIEW_PrintfHost()` are shown in both the **Event List** and the **Terminal** window. When the target system is running FreeRTOS, the **Event List** will also record when tasks are run and blocked.

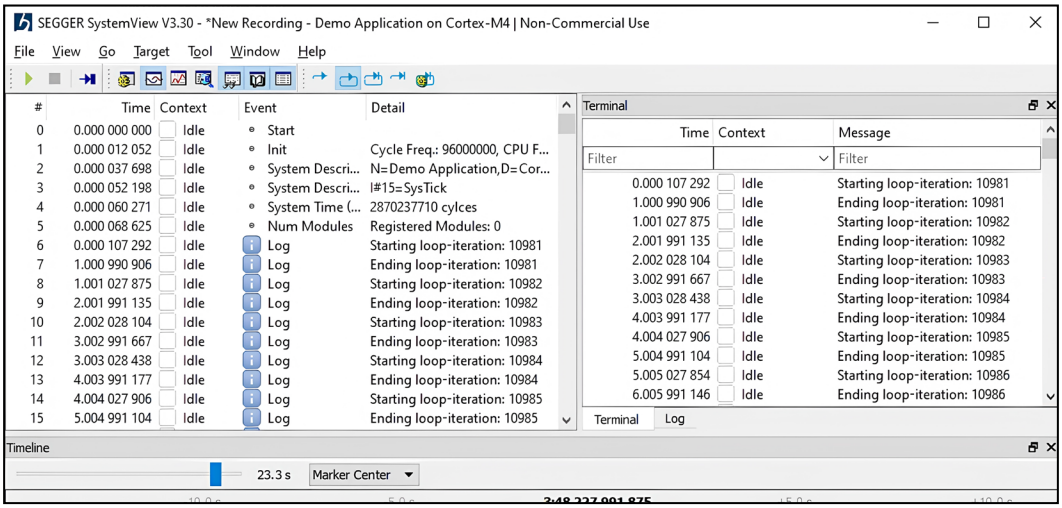


Figure 3.15 : SystemView

To set-up SystemView capabilities in the target program, a SystemView-API is called: `SEGGER_SYSVIEW_Conf()`. The API-call can be seen in the `Chapter_03` example-program, in `main.c`. The SystemView Recorder should not be started until after `SEGGER_SYSVIEW_Conf()` has been called. Later in the book, when FreeRTOS is being used, SystemView Recorder should not be started until after the target-system runs `vTaskStartScheduler()`.



For the SystemView screenshot in *Figure 3.15*, the loop-iteration counts are probably different than what you'll see, i.e., bigger or smaller values.

In general, for the tool snapshots in the book, there may be minor differences with what you'll see in running the tools. There are a variety of causes, such as different delays in starting SystemView.

In some cases, there can be minor differences between the example program in the code-repo, and the example program that is shown in the book and used for the screenshots.

This concludes the section on the debugging tools: J-Link, Ozone and SystemView. Their installation and use were presented. These tools will be used throughout the book.

Summary

This chapter presented the development-tools that we'll use throughout the book. The IDE we'll be using is the STM32CubeIDE, including its source-code editor, the build system, the system libraries, and the code generator.

Three SEGGER debugging tools are also used. A J-Link debug-probe is used, and it is made available by installing J-Link firmware on the ST-Link part of the dev-board. Ozone is used to load firmware on the MCU, and it provides source-code debugging. SystemView provides real-time monitoring of the firmware, and a means for viewing the application's debug-messages.

The book's example-programs were also presented, including how to install them for use in the IDE.

Appendix A provides a quick-reference for using the tools (i.e., a "cheat-sheet"). It summarizes the tool-use instructions from the present chapter.

In the next two chapters, we'll see how a super-loop program works, and how it can be created, using our dev-board and development-tools.

4

Understanding Super-Loops

The super-loop programming paradigm is typically one of the first programming methods that an embedded systems engineer will encounter. A program implemented with a super-loop has a single top-level loop that cycles through the various functions the system needs to perform. These simple `while` loops are easy to create and understand (when they are small). In RTOS systems, each task is very similar to a super-loop – the main difference is that RTOS systems can have multiple tasks, but super-loop systems only have a single super-loop.

In this chapter, we will take a closer look at super-loops and different ways of achieving a degree of concurrent operation with them. The primary purpose of this chapter is to provide the background information that is needed to understand the problems an RTOS solves and to understand RTOS features and capabilities.

This chapter presents super-loops from a design perspective, using pseudo-code. The next chapter shows a super-loop example program, and how it can be created on our dev-board.

The following topics will be covered in this chapter:

- Introducing super-loops
- Improving the super-loop with interrupts
- Adding a timer to the super-loop

Introducing super-loops

There is one common property that all embedded systems generally share – they typically don't have an exit point. Because of its nature, embedded code is generally expected to always be available – silently running in the background, taking care of housekeeping tasks, and ready for user input at any time. Unlike desktop environments that are meant to start and stop programs, there isn't anything for a microcontroller to do if it exits the `main()` function. If this happens, it is likely that the entire device has stopped functioning. For this reason, the `main()` function in an embedded system is designed to never return (or if it does, it usually provides error diagnostics for debugging purposes). Unlike application programs, which are started and stopped by their host OS, most embedded MCU-based applications start at power-on and end abruptly when the system is powered off. Because of this abrupt shutdown, embedded applications typically don't have any of the shutdown tasks normally associated with applications, such as freeing memory and resources.

The following code represents the basic idea of a super-loop. Take a look at this before moving on to the more detailed explanations:

```
void main ( void ) {
    initialize_system();
     while(1)
    {
        // Do useful stuff.
        func1();
        func2();
        func3();
        // But, don't return.
        // (Otherwise, where would we go? What would we do?)

    }
}
```

While extremely simple, the preceding code has a number of features worth pointing out. The `while` loop never returns (therefore the `main` function never exits.) The `while` loop goes on forever, executing the same three functions (this is intended). And, although the code may look simple, in a real-time system there are many things to consider in regard to timing, to ensure that the three functions don't violate any timing constraints.

The basic super-loop

This main loop that never returns is generally referred to as a ***super-loop***. It's always fun to think super because it has control over most things in the system – nothing gets done in the following diagram unless the super-loop makes it happen. This type of setup is perfect for very simple systems that need to perform just a few tasks that don't take a considerable amount of time. Basic super-loop structures are extremely easy to write and understand. If the problem you're trying to solve can be done with a simple super-loop, then use a simple super-loop. Here is the execution flow of the code presented previously – each function is called sequentially and the loop never exits:

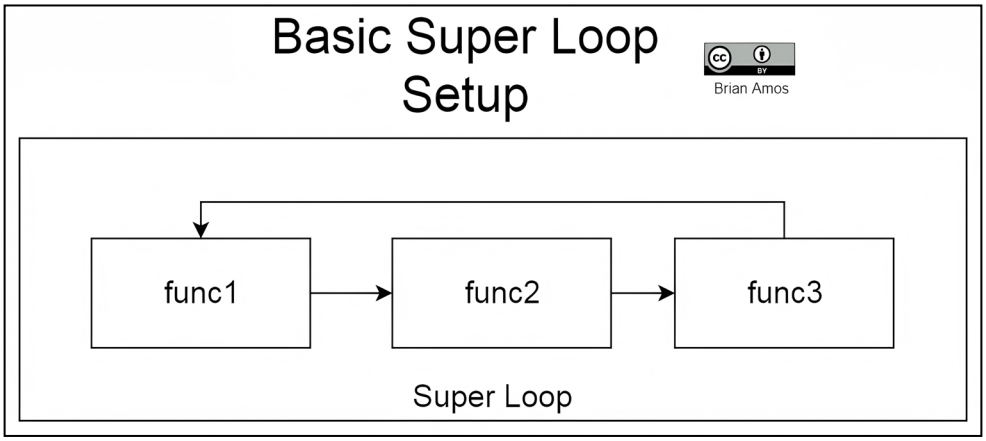


Figure 4.1: Flow chart of the example super-loop

In discussing super-loops, we'll need to define some terms. Each pass through a loop is a ***loop iteration***, or just ***iteration***. The ***iteration frequency*** is the rate at which iterations are processed. For a loop, the time it takes to process an iteration is referred to as the ***iteration execution time***, or alternatively, the ***iteration period***. A loop that is inside another loop is referred to as an ***inner loop***.

Typically, a super-loop iteration executes a set of functions, sequentially. A ***function's frequency*** specifies how often the function is run. The ***function's period*** is the time between when the function is started and the next time it is started. A function typically executes for just part of its period. For example, a function that reads a temperature sensor is called every 1 ms, but when the function is called, it runs for much less than 1 ms.

Now, let's have a look at what super-loop execution looks like in a real-time system and some of the drawbacks associated with this approach.

Super-loops in real-time systems

While super-loops are simple, they only work successfully if the iteration's execution time is less than the shortest period required for the functions. For example, if `func1` has to be run at least every X ms, then the time it takes to run all three functions must be X ms or less. There are many systems where this is the case and super-loops are a good solution.

In a super-loop, each function follows the preceding function. The functions are always called in the same order, and they are dependent on one another in meeting timing requirements. Any delay introduced by one function propagates to the next function and increases the time it takes that super-loop iteration to execute. An example of this can be seen in the following diagram. If `func1` takes 10 ms to execute the first three times through the loop, and then 100 ms the fourth time through, `func2` is going to be delayed by quite a bit relative to previous execution times:

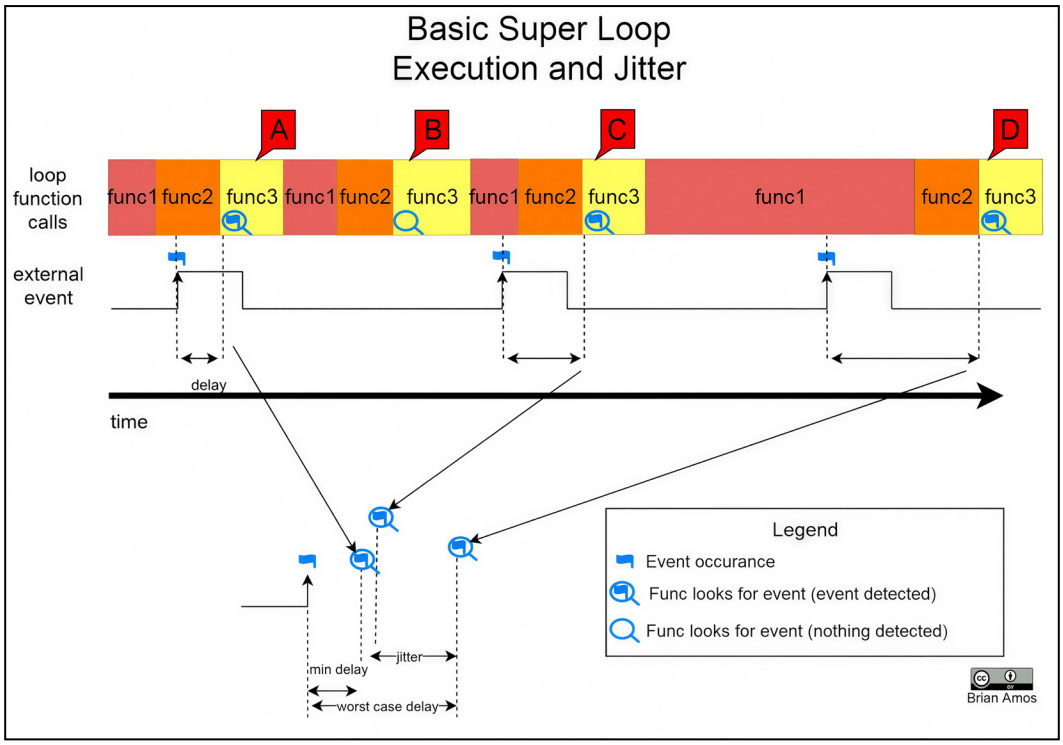


Figure 4.2: An example super-loop, showing the effects of variation in function duration

Let's take a look at this in a little more depth. In the preceding diagram, let's assume `func3` is responsible for periodically checking the state of an external-event signal (this is referred to as ***polling***). The frequency of how often `func3` checks the signal is dependent on how long `func1` and `func2` take to execute. In order for `func3` to detect every external event, it is necessary that the time between `func3` executions be shorter than the time between external events. A well-designed and responsive super-loop will typically execute very rapidly, checking for events more often than they occur. When an external event does occur, the loop doesn't detect the event until the next time `func3` executes.

In the figure, `func3` detects events at points A, C, and D. Notice that there is a delay between when the event occurs and when it is detected by `func3`. We'll refer to such a delay as the ***event-detection delay***, or simply, the ***delay***. At the bottom of the figure, a timeline shows when each event was detected, relative to when it occurred. For the three events, the timeline allows us to compare their event-detection delays. Note the delays were not consistent. The variation in the delays is referred to as ***jitter***.



In many super-loop-based systems, the iteration frequency is extremely high compared to the frequency of the events being monitored and controlled. We don't have enough room on the page to show a loop executing hundreds (or thousands) of iterations between detecting an event!

If a system has a known maximum amount of jitter in responding to an event, the system is considered to be ***deterministic***. That is, it will reliably respond to an event within a specified amount of time after that event occurs. Determinism is crucial for time-critical components in a real-time system because, without it, the system could fail to respond to important events in a timely manner.

The disadvantage of this super-loop design is that the loop must execute at the fastest frequency required for the polled events. This means that all other functions need to occur within very tight time constraints. In some simple cases, this is possible. In more complex systems, there may be functions whose execution times vary, and occasionally are longer than what is desired. Moreover, for functions with varying execution times, it may be difficult to determine the worst-case execution time. This leads to uncertainty about whether the super-loop will always meet its deadlines. There are ways to mitigate these problems, including the use of **Interrupt Service Routines** (ISRs), which we will explore next. And, variation in functions' execution times is one of the reasons for considering an RTOS for a real-time system.

Improving the super-loop with interrupts

Even though a basic super-loop can only step through functions sequentially, there are still ways to achieve concurrency and meet critical time-constraints. MCUs have some features to aid in responding to events in a timely fashion. This section will introduce those features and describe how they can be used in a super-loop-style program.

Introducing interrupts

As mentioned before, making the super-loop execute at the frequency of the fastest event may be difficult in complex systems. Also, polling for a single event can be wasteful in terms of CPU cycles and power use. And, polling may result in a system that cannot be responsive to anything else, which generally should be avoided. So then, how can we get a single-core processor to do things in parallel? Well, we can't–there's only one processor after all. But, when a processor is capable of running millions of instructions per second, it is possible to get the processor to perform things concurrently and appear to be performing things in parallel. A feature of MCUs that helps achieve this is some sort of interrupt-handling mechanism.

Interrupts provide signals to the MCU that allow it to jump out of executing one set of code and directly into another set of code called an **ISR**. The ISR is said to have higher **priority** than the code that it interrupts. After the ISR completes its execution, the MCU jumps back to the spot in the code it was executing when it was interrupted–as if the interrupt never happened! This is such a critical piece of functionality that ARM Cortex-M cores provide a standardized peripheral for it, called the **Nested Vector Interrupt Controller** (**NVIC**). The NVIC provides a common way of dealing with interrupts. The nested portion of this term signifies that even interrupts can be interrupted by other interrupts with a higher priority. This is quite convenient since it can allow us to minimize the amount of latency and jitter for the most time-critical pieces of the system.

So, how do interrupts fit into a super-loop in a way that better achieves the illusion of parallel operation? Answer – by using interrupts and ISRs to service events that need to be handled at a faster rate and/or with less jitter than the super-loop can achieve on its own. However, since the ISR interrupts the execution of the super-loop, it adds to the execution time and jitter of the super-loop. Therefore, the code inside an ISR is generally kept as short as possible. This is important for a few reasons. If the interrupt occurs very often and the ISR contains a lot of instructions, there is a chance that the ISR won't complete in time to service its next event. This could mean dropped data or missed event handling. Another reason to keep the ISR code short is that other interrupts may need to be serviced, and an ISR blocks other ISRs from running that are at equal or less priority.

Understanding ISRs and jitter

To better understand the use of ISRs, and how jitter is reduced, but not eliminated, let's look at a simple example. Let's assume we have an external **analog-to-digital converter** (**ADC**) signaling to an MCU that a measurement has been completed and the result is ready to be transferred to the MCU (refer to the hardware diagram shown here):

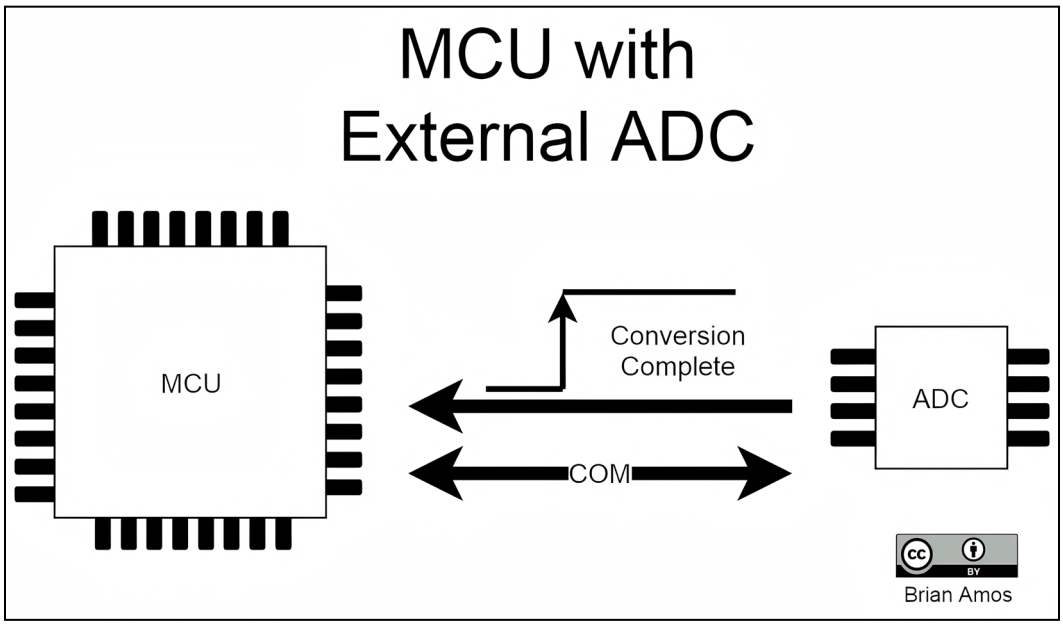


Figure 4.3: Exploring jitter, when using an ISR to process incoming ADC data

Let's also assume that the ADC conversion-complete signal is connected to a pin on the MCU. And, the MCU is configured to generate an interrupt on the rising edge of the signal. When the interrupt occurs, an ISR executes. The ISR uses a communication peripheral (COM in the diagram) to transfer the result from the ADC into the MCU.

Next, let's have a look at an interrupt's ISR calls, and the variation in the time between when an event occurs and when the ISR starts. The following diagram shows six different calls to an ISR. The ISRs are listed vertically, and the diagram shows when each started running, relative to its own start signal (the rising edge). The small amount of time between the rising edge of the signal and the beginning of the ISR is the **latency**. In the figure, the vertical dashed lines show the smallest and largest latency, among the ISRs. Notice that the latency is not the same for each ISR call. One way to measure jitter is the worst-case difference in this latency over many ISR calls.

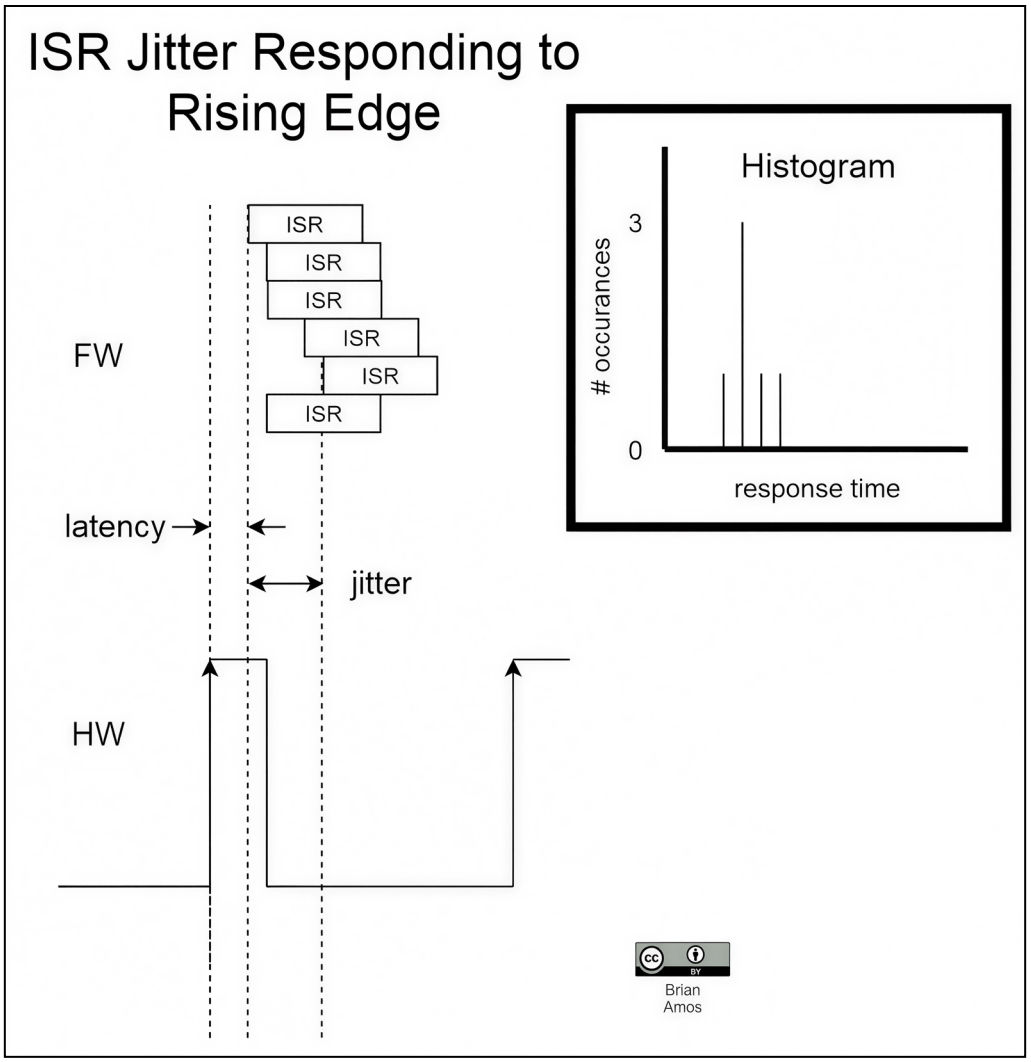


Figure 4.4: Jitter of an ISR when responding to a periodic external event

There are different ways to minimize latency and jitter for critical ISRs. In ARM Cortex-M-based MCUs, interrupt priorities are flexible – for a single interrupt source, its priority can be changed at runtime. The ability to dynamically reprioritize interrupts is one way of ensuring the most important parts of a system get the CPU when they need it, and therefore minimize latency and jitter. It is up to the designer to use such mechanisms in a way that meets the necessary latency, jitter, and execution time requirements of the real-time system.



ARM Cortex-M processors have a really useful feature called interrupt tail-chaining. This feature reduces latency when transitioning between ISRs. If the processor detects that an interrupt is about to exit, but another one is pending, the next ISR will be executed without the processor totally restoring the pre-interrupt state.

Interrupts and super-loops

For an ISR, one way of achieving minimal instructions and responsibility is to do the smallest amount of work possible inside the ISR, and for the ISR to set a flag that is checked by code running in the super-loop. When the flag is `true`, the super-loop will complete the processing for the interrupt. This way, the interrupt can be serviced as soon as possible, without the entire system being dedicated to waiting on the event. In the following diagram, notice how the interrupt is generated multiple times before finally being dealt with by `func3`.

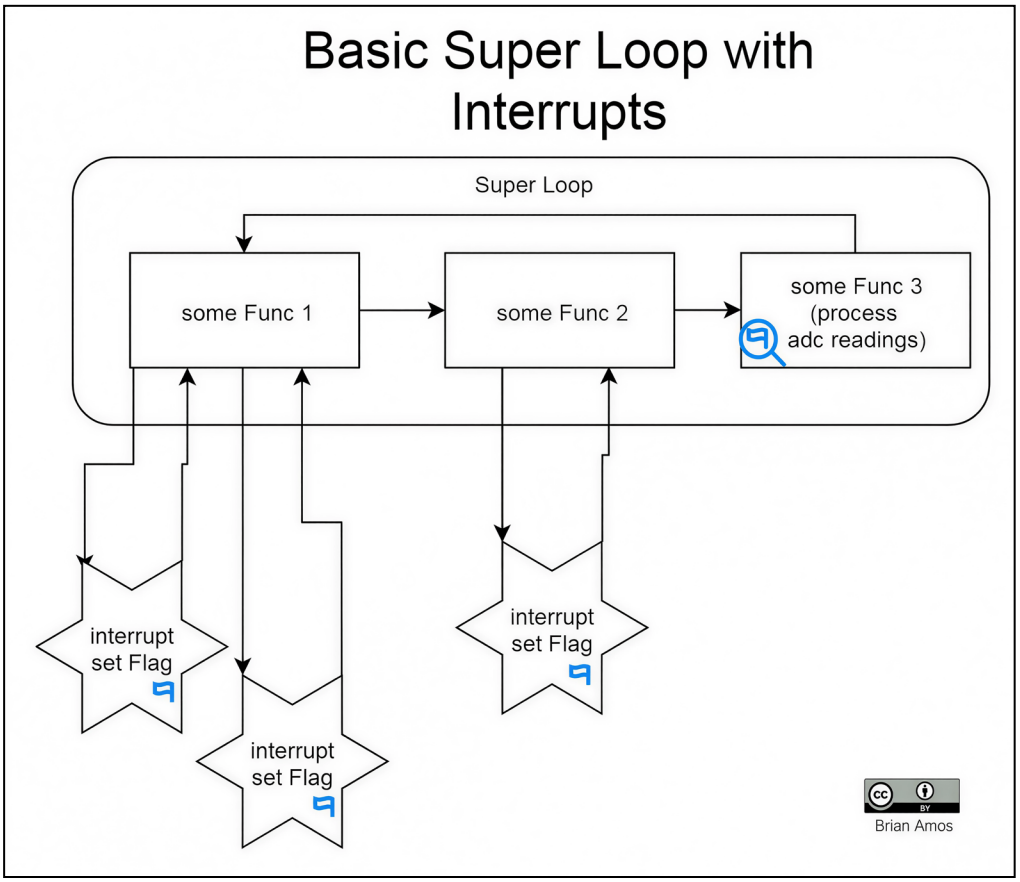


Figure 4.5: Setting a flag in an ISR, and processing its data later in a function

Depending on what exactly that interrupt is trying to achieve, it will typically take a value from the associated peripheral and push it into an array (or take a value from an array and feed it to the peripheral registers). In the case of our external ADC, the ISR (triggered every time the ADC performs a conversion) would go out to the ADC, transfer the digitized reading, and store it in RAM, setting a flag indicating that one or more values are ready for processing. This allows for the interrupt to be serviced multiple times without involving the higher-level code. In addition, when the super-loop and an ISR operate on the same data, it is necessary to ensure the operations do not overlap and conflict with each other. Two mechanisms for ensuring this are atomic instructions and critical sections. They will be discussed later in the book.

As we add more functions and ISRs to a super-loop, the interdependencies between them become increasingly complex. The next section presents algorithms that can be used to make super-loop processing more predictable and reduce the overall complexity.

Adding a timer to the super-loop

As mentioned, one of the super-loop's downsides is that the iterations' execution time can vary or be uncertain. For a super-loop iteration, how long the iteration takes depends on how long each function takes in the iteration. A slight modification is often made to the super-loop so that each iteration takes the same amount of time, and the iterations start at predictable time intervals:

```
// Start the super-loop's repeating timer
start_timer();
while(1)
{
    func1();
    func2();
    func3();
    // Poll the timer until its current period expires
    while( !timer_expired_flag );
}
```

Much of this super-loop looks just like the previous super-loop. However, after the functions have been executed, there is an inner `while` loop that waits for a timer to expire before allowing the functions to execute again. Upon the timer's expiration, a timer interrupt is generated, and the timer's ISR is called. The ISR sets a flag to `true`, indicating the timer expired (`timer_expired_flag`). When this flag is set to `true`, the super-loop ends the current iteration and starts the next iteration, where it executes `func1`, `func2`, and `func3` again. A repeating timer is used so that immediately after the timer expires, it restarts. The timer's period is set to the super-loop's intended period, for example, 1 ms. Overall, this method yields a super-loop with an iteration once per timer period.

Once the timed super-loop is created, you have just introduced a time deadline into the system–the super-loop period. And, once the super-loop has a periodic execution time, the total execution time of all functions must be less than the super-loop period. This isn't necessarily a new difficulty. Even an untimed super-loop requires that functions have a short enough duration to meet the time constraints of the system.

What we have effectively done here is create a **schedule**. We have scheduled the functions to execute once per each timer period. The untimed super-loop did not do this. Instead, it executed the functions as quickly as possible. This meant that the functions did not know how long it had been since they had executed before. However, in a timed super-loop, each function can count on being executed only once per timer period. Some functions may be written to take advantage of this knowledge, especially in regard to timing certain events.

(You may be wondering, what happens if executing the three functions takes longer than the intended super-loop period? Code could be added to detect and handle that condition. However, that's beyond the scope of our current focus.)

Now, it may seem counterintuitive to do nothing and wait for a timer to expire – isn't that *wasting* processor bandwidth?

Most real-time systems process events that occur at random times. For some real-time systems, especially those with communication stacks, the rate or number of events are configurable. An example is the number of TCP/IP connections allowed. So, the maximum rate of events can be known, but not when each event will occur. Therefore, it is often important for a real-time system to have some unused processor bandwidth, to handle peak demands. For a timed super-loop, even though the inner `while` loop uses processor bandwidth, it is able to yield to ISRs while waiting for the timer to expire.

Also, it could be argued that a super-loop that employs all of the processor bandwidth is actually more wasteful than a super-loop with an idle spin-loop. If the functions are running faster than is necessary to handle the systems events, then the functions aren't really performing any useful duties – they are simply running faster than is necessary.

So, doing nothing and waiting for a timer to expire isn't necessarily wasteful, as it provides some headroom to allow ISRs to execute while waiting for the next iteration. However, it does waste energy for the CPU to execute the `while` loop over and over. In low-powered battery systems, this waste of energy could be a big deal.

So, there is a mechanism that is typically added to the timed super-loop to conserve energy while waiting in the inner loop. That mechanism is the processor's `sleep` function, and it is called inside the inner while loop. This architecture is an important part of low-power systems, and it is described next.

Putting the timed super-loop to sleep

Most MCUs have special instructions that cause them to stop executing instructions. Some MCUs can stop executing instructions and put the processor into one of several different low-power modes. For our MCU (STM32F767ZI), the low-power modes include *sleep*, *stop*, and *standby*. (See the ST document *Managing low-power consumption on STM32F7 Series microcontrollers*, AN4749.)

The timed super-loop can take advantage of this mechanism to conserve power while it is waiting for the timer period to expire. We can see how this is done in the following code snippet, which is a modification of the timed super-loop's inner while loop:

```
    disable_interrupts();
    while( !timer_expired_flag )
    {
        sleep();
        enable_interrupts();
        disable_interrupts();
    }
    enable_interrupts();
```

Calls to three functions were added: `sleep`, `enable_interrupts`, and `disable_interrupts`. The details of how these functions work vary across MCUs. We'll describe how these functions work for our MCU.

The `sleep` function stops the CPU clock, and consequently, the CPU stops executing instructions. However, the MCU's peripherals are kept running, including the super-loop's timer. What ends the sleep mode is an interrupt occurring. In our timed super-loop, the `sleep` function is executed, and the super-loop timer will expire some time later. When the timer expires, it generates an interrupt, and that interrupt will end the sleep.

To more fully understand how the `sleep` function works, additional details are needed. Before entering the inner while loop, interrupts are disabled for the CPU. When interrupts are disabled, if a particular interrupt arrives, the interrupt's processing will be put in the *pending state*. When interrupts are later enabled, any pending interrupts will be put in the *active state*.

Putting an interrupt in the active state causes its ISR to run. (Putting an interrupt in the pending state does not cause its ISR to run.) (If multiple pending interrupts are put in the active state, their ISRs are run in priority order.)

When the `sleep` function runs, a pending interrupt will cause it to end. So, when sleep is started, if there are already pending interrupts, then the sleep will end immediately. After the sleep is started, if an interrupt arrives and is made pending, then the sleep will end.



In our MCU, `disable_interrupts()` is called before `sleep()`. For many other MCUs, their sleep-related functions work differently. A typical sequence is to call `enable_interrupts()` before `sleep()`. The sleep is ended by an interrupt that has arrived and is enabled. Also, the instruction after `enable_interrupts()` is always executed, and that pair of instructions cannot be interrupted by an ISR. An example of such an MCU is the TI MSP430FR4xx.

In the inner while loop, after the `sleep` function ends, interrupts are enabled. This allows the pending interrupts' ISRs to run. If the timer expired and its ISR was run, `timer_expired_flag` will be set to `true`. After the ISRs run, the inner while loop will resume, and interrupts will again be disabled. This completes the inner while loop iteration. Next, the while loop's condition is evaluated. If the timer's ISR ran, the timer-expired flag will be set to `true` and the loop will end. If the timer-expired flag is not set to `true`, then the sleep was ended by some other interrupt, and the loop will continue.

The `sleep` function is designed to be used with timers, and with the enabling and disabling of interrupts. In the code snippet, the `sleep` function is called in a loop. That way, if the sleep is ended by an interrupt other than the timer, that interrupt can be processed immediately and the sleep can resume.

Also, the disabling and enabling of interrupts is needed for the inner while loop to be able to run concurrently with the timer ISR. For example, the timer interrupt could occur after the while loop's condition is evaluated, but before the sleep is started. By disabling interrupts before evaluating the `while` loop condition, that timer interrupt would be made pending, and the sleep would end immediately. If interrupts were not disabled then, the timer interrupt's ISR would run immediately, and before the sleep. The sleep would then run until the next interrupt, which could be a full timer period.



There is a variation of a timed super-loop in which the processor is put to sleep for the whole loop period, and the functions are implemented in ISRs. Processors like our Cortex-M7 provide priority scheduling for ISRs, and that scheduling mechanism can be used to run the super-loop's functions concurrently, instead of just sequentially. For example, the interrupt for `func2` arrives and `func2` runs in the ISR. But, while the `func2` ISR is running, the interrupt for `func1` arrives, which has a higher priority. The `func1` ISR would run, preempting the `func2` ISR. When the func1 ISR completes, the `func2` ISR would resume.

An implementation of the timed super-loop is shown in the next chapter, in an example program that runs on our dev-board. As we'll see, although the super-loop design is simple, it can be challenging to implement correctly due to the concurrent processing by the super-loop and ISRs, and due to timing requirements, memory barrier requirements, debugger requirements, etc.

So, in the next chapter, we'll look at actual super-loop code. For the remainder of the present chapter, we'll continue using pseudo-code to explain the super-loop design.

Taking advantage of the timed super-loop

Now that we've added a timer to the super-loop, there are some ways we can structure the code in the functions to take advantage of the timer. Also, there are some things we need to watch out for. The timed super-loop has its own deadline—the super-loop's period. The super-loop's functions must be designed so that their total execution time never exceeds the super-loop's period.

For simple functions, this may be easy. But for complex functions, with many execution paths, it may be difficult to guarantee that the super-loop's iterations will never take longer than the intended period.

To flush this out, let's think about a function called by a super-loop, and this function writes two bytes to Flash memory. When writing to Flash memory, there is often a wait time that must be observed between writes. Often, this is in the order of several milliseconds (let's use 5 ms for this example). This wait time is typically longer than the super-loop period. A common value for the super-loop period is 1 ms, which we will assume for this example. So, let's think about the implications of the following code:

```
void func_write_two_bytes( uint8_t byte1, uint8_t byte2 )
{
write_Flash(byte1);
Wait_5ms();
```

```
write_Flash(byte2);
Wait_5ms();
}
```

How long does this code take to execute? At least 10 milliseconds, which means that it is at least 9 milliseconds longer than the super-loop period!

So, how can this function be structured to perform its duties without violating the 1-ms time constraint for the super-loop period? The solution is to execute the function in pieces, over at least 10 iterations of the super-loop. This is typically implemented by the use of a state machine.

Below is what that code could look like. The process of writing two bytes is broken up into 12 pieces, and each piece is executed in a super-loop iteration:

```
typedef enum {
    SM_WRITE_BYTE1,
    SM_WAIT_5MS_BYTE1,
    SM_WRITE_BYTE2,
    SM_WAIT_5MS_BYTE2,
    SM_DONE
} sm_state_t;

static struct {
    uint8_t byte1;
    uint8_t byte2;
    uint32_t wait_timer;
    sm_state_t current_state;
} sm;

const uint32_t sm_timer_init = 5;

void func_write_two_bytes_init_state_machine( uint8_t byte1, uint8_t
byte2)
{
    sm.byte1 = byte1;
    sm.byte2 = byte2;
    sm.current_state = SM_WRITE_BYTE1;
}
```

```
void func_write_two_bytes_state_machine( void )
{
    switch( sm.current_state )
    {
        case( SM_WRITE_BYTE1 ):
            write_Flash( sm.byte1 );
            sm.current_state = SM_WAIT_5MS_BYTE1;
            sm.wait_timer = sm_timer_init;
            break;
        case( SM_WAIT_5MS_BYTE1 ):
            sm.wait_timer -= 1;
            if( sm.wait_timer == 0 ) {
                sm.current_state = SM_WRITE_BYTE2;
            }
            break;
        case( SM_WRITE_BYTE2 ):
            write_Flash( sm.byte2 );
            sm.current_state = SM_WAIT_5MS_BYTE2;
            sm.wait_timer = sm_timer_init;
            break;
        case( SM_WAIT_5MS_BYTE2 ):
            sm.wait_timer -= 1;
            if( sm.wait_timer == 0 ) {
                sm.current_state = SM_DONE;
            }
            break;
        case( SM_DONE ):
            // Nothing to do, all done!
            break;
    } // End of switch statement
}

int main(void)
{
    ---
    sm.current_state = SM_DONE;
    data_to_write = 0;
```

```
    ---
    while(1)
    {
        ---
        if ( (sm.current_state == SM_DONE) && (data_to_write == 1) ) {
             func_write_two_bytes_init_state_machine(byteA, byteB);
            data_to_write = 0;
         }
        func_write_two_bytes_state_machine();
        ---
        while( !timer_expired_flag )
        {
            sleep
            ---
        }
    }
    ---
}
```

You can immediately see a trade-off in creating a function to meet the timed super-loop's time constraints -- the code gets longer and more complex! (Though, in this case, we also have the benefit of using the processor more efficiently, compared to having it be idle for 5 ms for each write.)

Before using the state machine to write data, the state machine must be initialized by calling `func_write_two_bytes_init_state_machine()`. In each super-loop iteration, this initialization function will be called if: a) there are two bytes to be written to Flash memory, and b) the state machine is currently not in the process of writing to Flash memory. The struct `sm` records the data needed to run the state machine. The data in `sm` includes the two bytes to be written to Flash memory, and the state machine's current state (`sm.current_state`). The initialization function configures `sm` so the state machine can start running. This includes setting the current state to the state machine's first state, which is `SM_WRITE_BYTE1`. The states are defined in the enum `sm_state_t`, and there are five in total.

The state machine is implemented by `func_write_two_bytes_state_machine()`. It is called in each iteration of the super-loop. In each call, a `case` statement is used to run the code for the current state. As mentioned, the first state is `SM_WRITE_BYTE1`. This state writes the first byte to Flash memory. It then updates `sm.current_state` to the next state, `SM_WAIT_5MS_BYTE1`.

The state `SM_WAIT_5MS_BYTE1` will be run for the next 5 iterations of the super-loop. This state's purpose is to provide a 5-ms delay between writing the first and second bytes to Flash memory. This state uses the counter `sm.wait_timer` to record how many more iterations there are for the state. One of the advantages of a timed super-loop is the ability to implement timed delays over multiple iterations.

The next two states are like the first two states. In the state `SM_WRITE_BYTE2`, the second byte is written to Flash memory. The state `SM_WAIT_5MS_BYTE2` runs for 5 super-loop iterations. It ensures there is at least a 5-ms delay before another byte is written to flash memory.

The last state is `SM_DONE`. It does nothing. It will be run on subsequent super-loop iterations if there is no data to be written to Flash memory.

The state machine's performance must meet the timed super-loop's time requirements. In particular, for each super-loop iteration, the iteration's worst-case execution time must be much less than 1 ms. This includes the processing done by the state machine and any additional functions run during the iteration. Most of the example state machine will meet that constraint without any problems. The only unknown is the execution time of the `write_Flash()` function. This function needs to be explored to make sure its execution time will never cause a super-loop iteration to exceed the 1-ms deadline.

Timing-related performance analysis

The best way to explore the execution time of a function like `write_Flash()` is to review its code. Unfortunately, since this is a pseudo-code example, we can't do that here. But it is good practice to consider every function your code executes to make sure they do not cause your code to violate timing constraints. Such code reviews provide the basis for understanding system performance. However, code reviews have inherent limitations, so performance testing is also needed.

There is an adage for working with dangerous systems, such as high-voltage power: *it's what you don't know that will kill you*. Similarly, for the embedded-system developer: *it's what you don't know that will "kill" your system*, i.e., cause unexpected bugs and performance problems. In embedded systems, the MCU and peripherals can be very complex and have a myriad of obscure effects and requirements. Code reviews can find the types of problems you already know about. Conversely, testing can find the types of problems you don't already know about.

With code, some functions are easy to review, especially functions the reviewer has written themself. However, for functions in a library, the source code might not be readily available or discernable. Also, there can be unknown system effects that do not get discovered in a code review. For instance, some compiler-generated code can be unexpectedly slow, as with division ("/") when the CPU does not have a division operator. Therefore, to ensure timing requirements are met, it is often necessary to measure a function's actual execution time, under various conditions.

Two common ways of measuring a function's execution time are (a) using a debugger's profiler, and (b) toggling I/O pins. A profiler can be used to measure a function's average and maximum execution time. Toggling an I/O pin can be as simple as adding two lines of code to the firmware. Looking at the pins on an oscilloscope can yield much insight into the execution sequence and timing of particular code.

We can measure the execution time of the example's `write_Flash()` function by setting an I/O pin high before calling the function and setting the pin low after calling the function:

```
set_io_pin( 1 );
write_Flash();
set_io_pin( 0 );
```

We can determine how long the `write_Flash()` function executes by connecting an oscilloscope probe to the I/O pin and measuring the pulse width of the signal on the pin. Toggling an I/O pin will be shown in the next chapter, with the chapter's example program and our dev-board.

Timed super-loops aid in meeting time constraints, and they allow the programmer to structure the code in ways that conserve processor bandwidth and that enable functions to work together. The super-loop and the timed super-loop are foundational architectures of bare-metal programming. While being both useful and commonly used, understanding them is also helpful in understanding how an RTOS works, and why one would choose to move from relatively simple super-loops to a more complex RTOS.

Summary

We covered quite a few concepts in this chapter in relation to bare-metal programming with super-loops. At this point, you should have a good understanding of how super-loops can be combined with interrupts to provide parallel processing to keep a system responsive, without the use of an RTOS. We introduced a timed super-loop that uses a `sleep` instruction and a timer to perform duties at periodic intervals. A state machine was introduced to help break up software duties into shorter segments in order to better share the processor with other functions.

The primary purpose of this chapter was to provide the context for understanding the problems an RTOS solves, and for understanding RTOS features and capabilities. The super-loop predates RTOSes, and it is still widely used. This chapter showed some of the super-loop's limitations in meeting timing requirements, and how even simple attempts to enhance the timing capabilities can be surprisingly complex.

In the next chapter, we present a super-loop example program and show how to create it on our dev-board.

5

Implementing the Super-Loop

This chapter presents an example program that implements a timed super-loop. This chapter also shows you how to create the program using the dev-board's IDE. The timed super-loop's design was presented in the prior chapter.

The primary purpose of this chapter is to provide background information on embedded-systems programming, for programmers who are new to embedded-systems, or new to the MCU and IDE. The example-program is provided in source code, and it can be built with the IDE, and run on the dev-board. The example-program is explained, and also, tutorial information is provided for those new to embedded-systems. Also, a tutorial is provided on how to create the example-program from scratch. Experienced embedded-systems programmers may want to just skim this chapter, or skim its tutorial information.

More broadly, the present chapter provides a general understanding of how all of the book's example-programs were created, and how to further research the code, such as the system API calls.

An additional purpose of this chapter is to show that although the timed super-loop is conceptually simple, the implementation is relatively complex, due to a number of factors, which will be presented.

The following topics will be covered in this chapter:

- The super-loop example-program
- Learning about system APIs and hardware features
- How to develop the super-loop program

The super-loop example-program

The present chapter's example-program is a timed super-loop. (We'll refer to it here as simply *the example-program*.) The timed super-loop was described in the prior chapter, in pseudo-code format, and it is shown below. The example-program implements that pseudo-code. This section explains how the example-program works, and how to run it on the dev-board. How to create the program will be described later in the chapter.

```
// Start the super-loop's repeating timer
start_timer();
while(1)
{
    func1();
    func2();
    func3();

    // Sleep until the timer's period expires
    disable_interrupts();
    while( !timer_expired_flag )
    {
        sleep();
        enable_interrupts();
        disable_interrupts();
    }
    enable_interrupts();
}
```

The super-loop example-program is described in the following sections:

- Technical requirements
- How the example-program works
- Design and testing considerations
- Running the example-program

Technical requirements

To carry out the exercises in this chapter, you will need the following:

- A Nucleo F767 development board
- A micro USB cable

- STM32CubeIDE
- SEGGER J-Link, Ozone, and SystemView
- The *Chapter 5* example-program
- The reference documents listed in *Appendix B*, up to the present chapter

For the system software and hardware mentioned in this chapter, the documentation for them is listed in *Appendix B*. In this chapter, when referencing manuals, abbreviated titles are used for brevity, and the full titles can be found in *Appendix B*.

When working with software for an MCU, it is often important to have the related documentation close at hand, such as datasheets, user guides, and reference manuals. This is one way that developing software for an MCU differs from developing software for a general-purpose operating-system. For example, while the system APIs provided by the MCU-vendor often hide the details of a peripheral register-set, there is often the need to understand which peripherals the MCU has, and also in what ways the peripherals may be configured.

How the example-program works

How the example-program works is described in the following sections:

- Introducing the example-program
- Super-loop initialization
- The super-loop

Introducing the example-program

The chapter's example-program is in the IDE workspace `packtBookRTOS2E`, in the `Chapter_05` project. The example-program's source files are mostly under `Core/Src`. A debug build-configuration is provided.

The super-loop itself is in the source file `Core/Src/main.c`, in `main()`. Open `main.c` in the IDE. In `main()`, the super-loop is `while(1)`. The *super-loop initialization* is the code in `main()` before `while(1)`.

In `main.c`, most of the code was generated by STM32CubeIDE. The IDE has an interface for specifying the MCU to be used and for specifying the MCU peripherals to be used. For the example-program, that interface was used to specify the MCU's clock sources and clock rates, and to specify the use of the MCU's **random number generator** (**RNG**). The IDE created `main.c`, and it generated code in `main.c` to configure and start the specified clock and peripheral. How to use the IDE to specify peripherals and generate code will be shown later in the present chapter.

In describing the book's example-programs, only the parts that need to be explained are shown in the book. The book's program descriptions assume the readers are also studying the example-program's source code themselves.

The first part of `main()` is shown below. Omitted lines are marked as "---".

The generated code has sections where the programmer can add code. Each of those sections is delineated by a pair of line-comments. One such section is shown below. It starts with the line-comment "`/* USER CODE BEGIN 1 */`" and ends with "`/* USER CODE END 1 */`". The code after that section was generated by the IDE. In general, the code outside of the user-code sections was generated by the IDE.

It's important to keep user-code in the designated sections. If the hardware configuration is changed, the code has to be re-generated. However, the code in those designated sections will not be modified.

```
int main(void)
{
  /* USER CODE BEGIN 1 */
    // Super-loop initialization
    // Indicates whether debug-messages are to be generated
    const uint8_t debug_generateMessages = 1;
---
    // Disable interrupts during initialization, before the super-loop
    __disable_irq();
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/
  /* Reset of all peripherals, Initializes … the Systick. */
  HAL_Init();
---
  /* Configure the system clock */
  SystemClock_Config();
---
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_RNG_Init();
```

Super-loop initialization

The code shown above is the first part of the super-loop initialization. The user-code declares variables and disables interrupts. In our experience, it's good practice to disable interrupts during super-loop initialization, as possible. This avoids complications from the super-loop timer's ISR running before the rest of the code is fully initialized. In the code above, the code that is after the user-code was generated by the IDE.

`__disable_irq()` disables all interrupts, except for a few essential system interrupts (NMI, Hardfault, and Reset). `__disable_irq` is an API in the CMSIS library. That library is provided by ARM for common uses of ARM processors. The CMSIS APIs make it easier to use the processor, and they help to make the code portable across different implementations of the processor. The CMSIS documentation can be found on the ARM web page "CMSIS-Core support for Cortex-M processor-based devices" (`https://arm-software.github.io/CMSIS_5/latest/Core/html/index.html`).

Continuing with the super-loop initialization, `HAL_Init()` performs initialization for the **Hardware Abstraction Layer** (**HAL**) library. The HAL library is provided by ST, and the library's APIs are used to operate the MCU's peripherals and CPU, e.g., clocks, timers, ADCs, UARTs, flash memory, etc. When HAL APIs are used, `HAL_Init` must be called first among the HAL APIs. By convention, APIs with the prefix *HAL_* are from the HAL library. Our MCU's HAL library is documented in the ST user-manual *Description of STM32F7 HAL and low-layer drivers*.

For our timed super-loop, the timer that will be used is the *SysTick timer* (also known as simply *SysTick*). One of `HAL_Init`'s functions is to configure the SysTick timer. The HAL library also provides the ISR for the SysTick timer. The ISR is provided in source-code format, and user-code can be added to it, as will be shown.

SysTick is the "System Tick Timer." It is a special-purpose timer that is part of the Cortex-M7 CPU. (In contrast, general-purpose timers are MCU peripherals and not part of the CPU.) One of the SysTick-timer's typical uses is as a *tick timer*, which generates an interrupt at a particular rate (e.g., 1,000 Hz), and the SysTick ISR is run for each of those interrupts. For instance, FreeRTOS can use the SysTick timer to periodically run its scheduler, e.g., every 1 ms. And, HAL can use the SysTick timer to generate an interrupt every 1 ms, for use by HAL's library functions.

For example, `HAL_delay` provides a program delay, specified in milliseconds, and internally, `HAL_delay` uses the SysTick timer. (When FreeRTOS and HAL are both used, usually only one of them uses the SysTick timer.)

The SysTick timer is used for the super-loop's timer because the SysTick timer is easy to use. `HAL_init` configures and starts the timer, and user-code can easily be added to the ISR. A limitation of using the SysTick timer is that, practically, our timed super-loop can only have a 1 ms period since HAL is being used. A production super-loop would need more flexibility in setting the period. An alternative to using the SysTick timer is to use one of the MCU's peripheral timers (e.g., the general-purpose timer TIM3).

Continuing with the super-loop initialization, `SystemClock_Config()` configures and starts the clocks used by the CPU and peripherals. The function `SystemClock_Config` is in `main.c`, and it is generated by the IDE. The function uses HAL APIs to configure and activate the specified clocks.

The super-loop initialization also makes calls to `MX_GPIO_Init()` and `MX_RNG_Init()`. Functions with the name `MX_<Peripheral>_Init` are generated by the IDE, and they are in `main.c`. These functions initialize the peripheral named `<Peripheral>`. The function `MX_GPIO_Init()` initializes the GPIO pins. Two of these GPIO pins are used in the super-loop, to activate the dev-board's green and blue LEDs. The function `MX_RNG_Init()` initializes the MCU's RNG, which is also used in the example-program.

Each of these functions generated by the IDE has calls to various HAL APIs, and these HAL APIs are used to operate MCU peripherals. In `SystemClock_Config`, HAL APIs are called to initialize the clocks. These APIs have names of the form "`HAL_RCC...Config`". *RCC* is the MCU peripheral *Reset and Clock Control*. It handles configuration of the MCU's clock tree. For many MCU peripherals, its HAL APIs are named "`HAL_<Peripheral>_...`". Examples are `HAL_GPIO_Init` and `HAL_RNG_Init`, called in `MX_GPIO_Init` and `MX_RNG_Init`, respectively.

SysTick is specified in ST's manual *STM32F7... Cortex-M7 processor programming manual*. Further description of the MCU's clocks and GPIOs are beyond the scope of the present book. The book *Mastering STM32* has tutorials on these topics, as well as tutorials on SysTick and HAL.

Next, we'll look at the second part of the super-loop initialization. It is shown below, and it is all user code.

```
  /* USER CODE BEGIN 2 */

    if (debug_generateMessages == 1){
        SEGGER_SYSVIEW_Conf();
    }
    HAL_DBGMCU_EnableDBGSleepMode();
---
```

```
    // Synchronize with SystemView Recorder and SysTick timer
    __enable_irq();
    __ISB();

    if (debug_generateMessages == 1){
        while(SEGGER_SYSVIEW_IsStarted()==0);
        SEGGER_SYSVIEW_PrintfHost("Super-loop starting...\n");
    }

    // Synchronize super-loop's first iteration with SysTick-timer's
period
    timerDone = 0;
    while(timerDone==0);

    __disable_irq();

  /* USER CODE END 2 */
```

The flag `debug_generateMessages` is a `const` that specifies whether SystemView debug-messages should be generated. `debug_generateMessages` is declared at the beginning of `main()`. Set it to `1` to generate debug-messages, and `0` otherwise.

`SEGGER_SYSVIEW_Conf()` configures SystemView so we can send debug messages to the development computer.

`HAL_DBGMCU_EnableDBGSleepMode()` enables debugging during sleep mode. For the timed super-loop, one of its distinctive features is putting the CPU in sleep mode. By default, a debug connection is lost if the MCU is put in sleep mode. So, `HAL_DBGMCU_EnableDBGSleepMode` is called here, as we'll be running the program in debug mode.

For the remaining initialization code, interrupts have to be enabled. `__enable_irq()` enables interrupts. It must be followed by a call to `__ISB`, which is a *memory barrier instruction*. Both of those APIs are from the CMSIS library. Here, `__enable_irq` enables the interrupts that were disabled by `__disable_irq`. A call to `__ISB` is needed after `__enable_IRQ` due to the processor's architectural requirements. `__ISB` ensures that any pending interrupts are recognized before the next instruction, i.e., pending interrupts are made active. More information is in the ARM document, *ARM Cortex-M Programming Guide to Memory Barrier Instructions*.

Debug messages are sent to the SystemView app on the development computer. To view the debug messages, the app must be started, as well as the app's Record mode. `SEGGER_SYSVIEW_IsStarted()` indicates whether the app's Record mode has started. The first while-loop here will spin until the app's Record mode has started.

The next while-loop spins until the SysTick timer expires. This ensures the super-loop's first iteration starts very close to the start of a timer period. Our timed super-loop depends on that condition for the correct sleep duration.

When the SysTick timer expires, the timer's ISR sets the `timerDone` flag to `1` (`true`). It's necessary to set `timerDone` to `0` (`false`) before the while-loop, in case the super-loop initialization takes more than 1 ms, though that is very unlikely.

The SysTick ISR's source-code is generated by the IDE. The ISR is in `Core/Src/stm32f7xx_it.c`, and the ISR is shown below. In the ISR's user-code section, `timerDone` is set to 1 (true). The ISR and `main()` use `timerDone` concurrently. Such concurrent access is possible with Cortex-M processors because its reads and writes to `uint8_t` variables are atomic. And, `timerDone` is a `uint8_t` variable. (More information on atomic memory-access is in the ARM document *ARMv7-M Architecture Reference Manual*, in the section *A3.5.3, Atomicity in the ARM architecture*.)

```
void SysTick_Handler(void)
{
---
  HAL_IncTick();
  /* USER CODE BEGIN SysTick_IRQn 1 */
  timerDone = 1;
  /* USER CODE END SysTick_IRQn 1 */
}
```

The super-loop's initialization ends with `__disable_irq()`. As we'll see next, the super-loop's iterations need to be started with interrupts disabled.

The super-loop

The first part of the super-loop itself is shown below.

```
while (1)
{
    __enable_irq();
    __ISB();
```

```
    debug_superLoopIterationCount++;
    timerDone = 0;

    // Generate a delay of random length, to simulate functions being run
    HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET);
    doStuffTime_us = getRandomNumberInRange(200,600);
    delay_us(doStuffTime_us);
    HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_RESET);

    if ( timerDone == 1 ) {
        debug_timerDoneBeforeSleepCount++;
        HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
        if (debug_generateMessages == 1){
            SEGGER_SYSVIEW_PrintfHost(
                "Iteration: %u, debug_timerDoneBeforeSleepCount: %u\n",
                debug_superLoopIterationCount,
                debug_timerDoneBeforeSleepCount);
        }
    }
```

The super-loop is designed so that when an iteration starts, the following conditions hold:

- Interrupts are disabled
- The SysTick-timer's period ended very recently, and the ISR set `timerDone` to 1 (true)
- As a corollary, the SysTick-timer's period started very recently

The super-loop is also designed so that its iterations have a fixed-length period. In this case, it's 1 ms. The iterations' period is implemented using the SysTick timer, which is configured with a 1 ms period.

The iteration's first instruction is `__enable_irq()`. Interrupts must be enabled as the iteration's primary processing depends on pending interrupts having been processed. Next, `timerDone` is set to 0 (false), so it can be examined later to determine if the iteration's primary-processing took too long.

The super-loop's purpose is to periodically run a set of functions that perform the system's primary processing. However, instead of running a set of functions here, they are simulated by creating a random delay between 200 and 600 microseconds.

`GetRandomNumberInRange` is a user-written function in `main.c`. That is, it's written by the book's authors, and it's part of the example-program. `GetRandomNumberInRange` is not shown here, and readers are encouraged to peruse it in `main.c`.

`getRandomNumberInRange` calls the MCU's RNG to get a 32-bit random number. The function then returns a random number in the specified range. Here the range is 200 to 600.

Information on the MCU's RNG can be found in the MCU's reference manual. Also, the HAL user-manual documents a number of RNG-related APIs. However, those sources do not provide tutorial information on using the RNG. Fortunately, ST provides an example-program that uses the RNG. Further, that RNG example-program runs on our dev-board, and it uses the HAL APIs for the RNG. `getRandomNumberInRange` was modeled after that RNG example-program. Also, the super-loop's `main.c` has code to initialize the RNG, and that code was generated by the IDE. That generated code is consistent with how the RNG example-program initializes the RNG. How to find the RNG example-program is specified in *Appendix B* (see *STM32Cube MCU Package examples...*).

`delay_us()` creates a delay for the specified number of microseconds. It's a user-written inline-function. It's defined in `Core/Inc/systick_timer_us_delay.h`. We'll use the function without an explanation of how it works. (If you'd like to know more, the function's comments describe how it works.)

`HAL_GPIO_WritePin()` is used to turn the dev-board's green LED on, then after the delay, turn it off. This is a user LED, and it is at the top center of the dev-board's MCU part. The LED serves two purposes. It visually shows when the system is running. (The LED appears solid green.) Also, the LED's GPIO pins can be connected to an oscilloscope, to measure the delay and the iteration's periods. This is shown in the upcoming section, "How to develop the super-loop program".

After the simulated functions, a condition tests if `timerDone` is 1 (true). If so, the iteration's primary-processing took longer than the timer's period, which is a timing error. The error is recorded via SystemView, and the blue LED is turned on. It's possible that the timer period could end after the condition is evaluated, and before the sleep. However, that is not considered an error since the iteration would be completed relatively soon (i.e., relative to a 1 ms iteration period). (The sleep is in the second part of the super-loop, and it will be described shortly.)

The example-program was designed to demonstrate a super-loop that meets its timing requirements. So, the call to `delay_us` was limited to 600 microseconds. In testing the program, timing errors did not occur.

In considering timing requirements, recall that the time it takes to process a loop iteration is referred to as the ***iteration execution-time***, or alternatively, the ***iteration period***. In the timed super-loop, we consider the time spent sleeping to be part of the iteration execution-time. (Sleep is an instruction that the processor executes, though in a low-power mode.)

When creating or modifying a timed super-loop, an embedded-systems programmer must be mindful of how long the non-sleep processing will take for an iteration. This includes the primary-processing and ISRs (aside from the super-loop timer's ISR). These ISRs could run at any time during the iteration, including during sleep. The ***super-loop's processor utilization*** is the time spent not sleeping, divided by the iteration period. For the example-program, the super-loop's processor utilization is a minimum of 20%, a maximum of 60%, and a mean of 40%. These measures assume no ISRs run during the call to `delay_us`, and that the super-loop's program-structure, such as `while(1)`, is negligible.

The second part of the super-loop is shown below. Its purpose is to sleep until the SysTick timer expires. This code implements the pseudo-code for the super-loop's inner while-loop. How that pseudo-code works was explained in the prior chapter, so it's not explained here.

```
    __disable_irq();

    // Sleep until the timer expires for the super-loop iteration
    while (timerDone == 0) {

        // Sleep until there is a pending interrupt
        HAL_PWR_EnterSLEEPMode(0, PWR_SLEEPENTRY_WFI);

        // Enable interrupts, so that pending interrupts' ISRs can run
        __enable_irq();
        __ISB();

        __disable_irq();
    }
```

`HAL_PWR_EnterSLEEPMode()` starts sleep mode. From the API's name, we see it's from the HAL library, and it operates the MCUs `PWR` (Power Controller) component. From the MCU's HAL user-manual, we see that the API's first parameter isn't used. The second parameter specifies to enter `SLEEP` mode with the `WFI` instruction. Describing the `WFI` instruction is beyond the scope of the book.

For additional information on `HAL_PWR_EnterSLEEPMode()`, the specification of the MCU's Power Controller component can be found in the MCU's reference manual. Specification of the CPU's Power Management component, and the `WFI` instruction, can be found in the CPU's *Generic User Guide* and its *Architecture Reference Manual*. Tutorials on these topics can be found in the books *Mastering STM32 and The Definitive Guide to …Cortex-M4 Processors*.

When the super-loop's inner while-loop completes, the following conditions hold:

- `timerDone` is 1 (true)
- Interrupts are disabled
- The SysTick-timer's period just started

This assumes the super-loop is operating as intended, and in particular, that the SysTick timer did not expire before the inner while-loop.

This concludes our presentation of the code for the super-loop example-program.

Design and testing considerations

This section addresses super-loop design issues related to timing requirements, and also, issues related to testing and debugging.

Timing resilience

How can we determine whether a timed super-loop's timing requirements will be met? The processes and techniques for determining that can be referred to as *timing assurance*. One aspect of timing assurance was described in the prior chapter: timing-related performance analysis. This included static code-analysis and testing techniques.

Another aspect of timing assurance is how resilient the algorithm and implementation are to timing-related problems. The example timed-super-loop is intentionally simple, and it assumes two timing-related conditions are always met: 1) an iteration and the timer period start very close to each other, and 2) the iteration period is always 1 ms, or very close to it. If those conditions are not met, the system's timing and its future ability to meet timing requirements are not specified (for the example-program). Additional analysis and testing could be performed to better understand what happens then; however, determining that precisely can be non-trivial.

A better approach would be to design the super-loop to be more resilient to timing-related problems. This would include mechanisms for detecting and recovering from timing-related problems, and also, designing the system so that its behavior will be more predictable when timing requirements are not met.

The example-program does include code for detecting and reporting when the timer period ends before the sleep mode. However, once the timing conditions are not met, the super-loop's future behavior, relative to the timing requirements, is not specified (for the example-program).

Testing and debugging

This section further describes the example super-loop's testing and debugging features. The timed super-loop presented here is intended for learning. So, a debug version is provided, and not a release version.

One challenge in testing and debugging the timed super-loop is that the testing and debugging tools can affect the system's timing, and cause the timing requirements to not be met. This is especially a problem with `SEGGER_SYSVIEW_PrintfHost`. From our testing, calling it in the super-loop appears to take over 200 microseconds, and sometimes significantly more. Thus, `SEGGER_SYSVIEW_PrintfHost` is only used here to report detected problems, and not to display ongoing operational information. Also, SystemView use can be turned off via a configuration `const` (`debug_generateMessages`).

Two alternatives to SystemView are also used, which have negligible effects on timing. Variables are used to store debugging information, and their contents can be viewed in debug mode. Also, GPIO pins are toggled. They turn on LEDs, and they can be viewed on an oscilloscope.

A potential problem with source-code debuggers is having timers continue to run when the processor is halted, e.g., when stepping through the code. With the SysTick timer, its counter does not decrement when the processor is halted in debug state. Fortunately, this is the behavior needed when debugging the super-loop. It is a CPU feature, and it's documented in the CPU's *Architecture Reference Manual*.

Debugging during low-power modes, like sleep, also presents challenges. By default, the debug connection is lost when the MCU is put in sleep mode. This is because the Cortex-M core is no longer clocked. `HAL_DBGMCU_EnableDBGSleepMode` is used to enable debugging during sleep mode. However, it causes sleep mode to use additional power. So, the function should not be used in release firmware. `HAL_DBGMCU_EnableDBGSleepMode` is a HAL function, and it sets the MCU register `DBGMCU_CR`. Additional information can be found in the book *Mastering STM32*, the HAL user manual, and the MCU reference manual.

For the debug probe, J-Link, its user guide lists restrictions on using the debugger while the target is in low-power mode. However, it's not clear whether all of those restrictions apply when `HAL_DBGMCU_EnableDBGSleepMode` is used. One of the restrictions that does apply is not using single-stepping to enter sleep mode. From our testing, when single-stepping into sleep mode, no sleep occurs, and sleep returns immediately, even when no interrupt is pending. Rather than single-stepping into the sleep function-call, a breakpoint can be set just after it, and the debugger can run to the breakpoint. The J-Link documentation also describes using an Ozone script for low-power debugging. However, it's not clear whether that script works with the J-Link version used on the dev-board. In our tests with the script, no effects on debugging were observed.

Running the example-program

The super-loop example-program can be run on the dev-board using the process presented in *Chapter 3*, *Introducing the Development Tools*: build the program; use Ozone to load the firmware on the MCU and start it; then start SystemView in Record mode. (That process is summarized in *Appendix A*.)

Here are additional instructions for running the example-program:

- During initialization, the program will spin until SystemView is started, in Record mode.
- Alternatively, the program's use of SystemView can be turned-off by setting `debug_generateMessages` to zero.
- When the program runs on the dev-board, the green user-LED will be on.
- In the debugger, single stepping into the sleep instruction is not supported.

After SystemView is opened and its Record mode is started, it should look something like this:

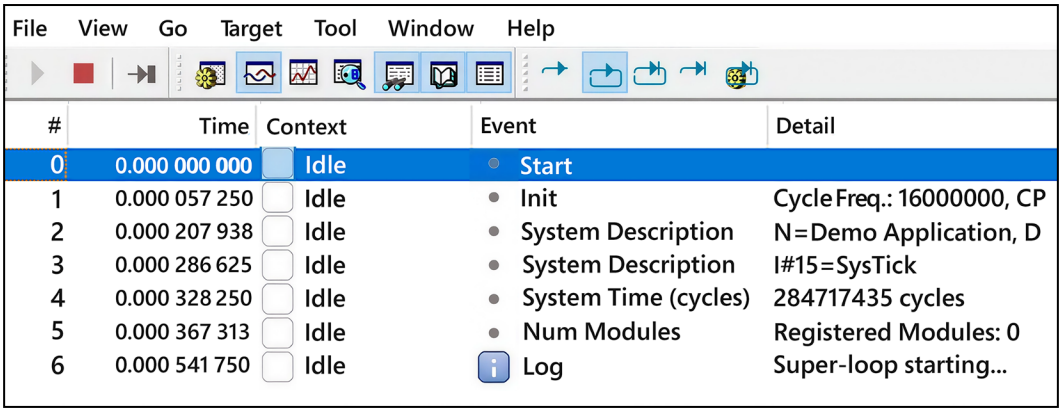


Figure 5.1: SystemView, in Record mode

In the super-loop example-program we encountered a number of system APIs, and also various hardware features such as the SysTick timer and the interrupt controller. While reading firmware source-code we often need additional information about the APIs and hardware features that are used. The next section shows how to find such information, using system documentation and the APIs' source-code.

Learning about system APIs and hardware features

The book's example-programs use a plethora of system APIs and hardware features. This section describes how to get information about them, and how to figure out what they do, how they work, and how to use them.

The book's example-programs frequently call system APIs, and the APIs are from several different sources, such as the HAL and CMSIS libraries. Also, the example-programs frequently use the hardware features of several different devices, including the MCU, CPU, dev-board, and debugger. Figuring out how these APIs and hardware-features work can be challenging, as the documentation consists of dozen of documents. Also, the libraries' documentation is often incomplete, so figuring out how to use an API often requires reading its source-code.

We'll examine two of the APIs that are used in the present chapter's example-program, and we'll show how we can learn more about these APIs.

HAL_DBGMCU_EnableDBGSleepMode

`HAL_DBGMCU_EnableDBGSleepMode` enables debugging during sleep mode. The prefix *HAL_* indicates it is an API in the Hardware Abstraction Layer (HAL) library, which is often referred to as simply *HAL*. Background information about HAL will be presented first, then information on how to learn about this API.

HAL is provided by ST, and the library's APIs are used to operate the MCU's peripherals and CPU, e.g., clocks, timers, ADCs, UARTs, Flash memory, etc. Each MCU family has its own HAL library. For instance, we're using an STM32F767ZI MCU, and it's in the STM32F7 MCU family. So, the book's example-programs use the STM32F7 HAL library. Each HAL library has its own user manual, and for our MCU it is *Description of STM32F7 HAL and low-layer drivers*.

When using the IDE to develop a project, the user specifies the MCU that is being used, and the IDE downloads the MCU's HAL library, in source-code. In the source-code that is generated by the IDE, much of it includes calls to HAL APIs, to configure and start the needed hardware features. An example is the call to `HAL_init` in the super-loop initialization. `HAL_init` provides initialization for using HAL APIs. `HAL_init` also configures and starts the SysTick timer, which is part of the CPU, and used by the timed super-loop. When building an IDE project, the HAL library is compiled and linked, as part of the created executable.

There are several benefits to HAL APIs. They can make it easier to use the MCU's features. Also, for different STM32 MCUs, their HAL libraries are consistent, which makes it easier to port code among the MCUs.

HAL can be challenging to use. In the HAL user manual, the API descriptions tend to be short. So, it is often necessary to read the HAL source-code. There, the code-comments tend to be sparse. In addition, a HAL API's source-code can be challenging to understand as it supports many use-cases, in addition to the one that is currently needed. For instance, a HAL API for configuring a timer may support various types of timers, and various ways of configuring the timers.

To learn more about `HAL_DBGMCU_EnableDBGSleepMode`, the obvious starting point is the HAL user manual. However, it only provides a brief statement about the API. And, currently, an Internet search returns very little information. There is an art to figuring out how to use APIs and hardware-features, when the documentation and online information are not sufficient. We'll follow one possible research path for finding more information about `HAL_DBGMCU_EnableDBGSleepMode`.

For HAL APIs, the second term in the API's name often specifies the MCU component being used. Here the second term is `DBGMCU`. Also, the API's source code can be examined. To do that in the IDE, in `main.c`, right-click on the API call, then select **Open Declaration**. In the API's source-code, we see the API just runs one instruction:

```
SET_BIT(DBGMCU->CR, DBGMCU_CR_DBG_SLEEP);
```

From that information it can be deduced that `DBGMCU` is probably an MCU component. By keeping system documentation in a searchable folder, a pdf search-tool can be used to find information related to `DBGMCU`. (For example, Adobe Acrobat Reader can search in all pdf files in a folder.) Searching the system documentation leads to the MCU's reference manual, and its section "44.16 MCU debug component (DBGMCU)". Skimming that section leads to the subsection "44.16.1 Debug support for low-power modes", as well as descriptions of the MCU's `DBG_SLEEP` feature. That feature is apparently being turned on via the `SET_BIT()` function.

What `SET_BIT()` does is fairly self-explanatory. For more information about `SET_BIT()`, the function-declaration and the parameters' declarations can be found by using the IDE. (Right-click on one of them and select **Open Declaration**). After opening the declaration for `DBGMCU`, we find it is the macro shown below.

```
#define DBGMCU    ((DBGMCU_TypeDef *) DBGMCU_BASE)
```

To find the declaration for `DBGMCU_BASE`, a different technique is needed. Select `DBGMCU_BASE`, then on the IDE toolbar click on **Search** : **Text** : **Project**. (Alternatively, on the IDE toolbar click on **Search** : **File**, and select **Scope** : **Enclosing project.)**

From the MCU reference manual and the API's source code, we can infer that `DBGMCU->CR` must be the MCU's register `DBGMCU_CR`. And, we can discover that `DBGMCU_CR_DBG_SLEEP` is a macro used to turn on the register's bit 0. (`DBGMCU_CR_DBG_SLEEP` is a parameter in the call to `SET_BIT()`.)

MCU components are typically operated via registers that are specific to the component. The register `DBGMCU_CR` is part of the `DBGMCU` component. It is 32 bits. Bit 0 is the `DBG_SLEEP` bit, which specifies whether sleep-mode is enabled. This is described in the MCU reference manual, in the section "44.16.4 DBGMCU_CR register".

The term "DBGMCU" can also be used to search for additional information, such as tutorials. ST's publication-search web page can be used, and of course, an Internet search engine. Though, in using them, we didn't find sources with additional information. The book *Mastering STM32* has a short, but useful, section on debugging in low-power modes.

The research path shown here is intended for someone with little experience. An experienced developer could skip steps, and get the information much faster. Next, we'll show how to learn about an API from the CMSIS library.

__disable_irq

`__disable_irq` disables all interrupts, except for a few essential system interrupts. As described earlier, `__disable_irq` is an API in the CMSIS library. That library is further described here, as well as a research path for figuring out what `__disable_irq` is and how it works.

In the IDE, in `main.c`, this API's source-code can be viewed by right-clicking on the API call, and selecting **Open Declaration**. In this case, the API contains a single instruction:

```
  __ASM volatile ("cpsid i" : : : "memory");
```

The API is in the file `Drivers/CMSIS/Include/cmsis_gcc.h`. The file-location reveals the API is in the CMSIS library.

The CMSIS library is provided by ARM for common uses of ARM processors. `__disable_irq` is in the CMSIS component *CMSIS-Core (Cortex-M)*, which is for ARM Cortex-M processors. "CMSIS-Core (Cortex-M)" is documented at the ARM web page "CMSIS-Core support for Cortex-M processor-based devices" (`https://arm-software.github.io/CMSIS_5/latest/Core/html/index.html`). Additional CMSIS documentation is listed in *Appendix B*.

CMSIS stands for *Cortex Microcontroller Software Interface Standard*. CMSIS defines interfaces for using ARM processors, and "CMSIS-Core (Cortex-M)" is one of many CMSIS components. CMSIS, as a whole, is large and complex. A description of it can be found in the book *The Definitive Guide to …Cortex-M4 Processors*.

As with HAL, the CMSIS documentation for APIs can be very short, and additional information is needed. The CMSIS documentation for `__disable_IRQ` is like that. It states that the API uses the instruction `CPSID i`, and sets the `PRIMASK` register.

Information on CPSID can be found by searching the recommended collection of MCU-related documents. Several sources are returned. The ARM processor's *Generic User Guide and the Architecture Reference Manual* both describe CPSID, and both should be consulted for complete information. In brief, we see that `CPSID i` disables interrupts by setting the `PRIMASK` register. The search-results also include the MCU's programming manual. Its information on CPSID is almost identical to what is in the ARM processor's *Generic User Guide*. In general, the ARM processor-documents specify the ARM standards, and the MCU processor-documents specify the parts of the ARM standard that are implemented.

Understanding `CPSID i` requires understanding how `PRIMASK` is used, and what it does. The document collection can be searched for `PRIMASK`, and the ARM *Generic User Guide* and the *Architecture Reference Manual* are returned.

However, perusing that PRIMASK documentation reveals that to understand it, one must first understand the processor's interrupt management component. The system manuals are typically not the best source of tutorials on complex system components. And, such a tutorial is beyond the scope of this book. Tutorials about that component can be found in the books *Mastering STM32* and *The Definitive Guide to …Cortex-M4 Processors*. A well-explained online tutorial is, *A Practical guide to ARM Cortex-M Exception Handling* (`https://interrupt.memfault.com/blog/arm-cortex-m-exceptions-and-nvic`).

When searching the document collection for the terms `__disable_IRQ`, `CPSID`, and `PRIMASK`, there is an additional document that is returned in all of the searches. It's the ARM Application Note, *ARM Cortex-M Programming Guide to Memory Barrier Instructions*. That document has a number of references to `__disable_IRQ`, so it is prudent to investigate them. However, investigation just reveals that calls to `__disable_irq` do not have to be followed by a call to a memory-barrier instruction, such as `__ISB`. In contrast, `__enable_irq` does have to be followed by a call to `__ISB`, as in our example-program. This investigation of `__disable_IRQ` is useful in dispelling any questions of whether `__disable_IRQ` must also be followed by a call to `__ISB`.

Of course, there are other research paths that could have been followed for `__disable_irq`, `CPSID` and `PRIMASK`. For example, we could have searched for those terms in the cited books. Also, an Internet search for those terms returns forum discussions and tutorials.

In general, a useful source of MCU-related tutorials are the example-programs that ST provides for MCUs and dev-boards. The examples include typical uses of MCU peripherals, such as timers and ADCs. How to get the ST example-programs for our dev-board is specified in *Appendix B* (see, "STM32Cube MCU Package examples…").

Having seen how the super-loop implementation works, we'll next see how it is created.

How to develop the super-loop program

This section shows how to develop the chapter's example-program, from scratch, using the IDE. This section is primarily intended for those who are new to embedded-systems development, or new to the IDE. The purpose is to provide context for understanding how the book's example-programs are created.

This section is optional. The book is about FreeRTOS, and it will be presented using example-programs provided with the book. Skill in creating such programs from scratch is not required.

How to work with the IDE and debugger was shown in *Chapter 3*, and basic familiarity with those tools is assumed here.

The following sections are presented:

- Create the super-loop program
- Install and configure the debugger software
- Build and run the firmware
- Using an oscilloscope to measure timing

Create the super-loop program

How to create the super-loop program is presented in these sections:

- Create the IDE project
- Configure the hardware
- Create the user-code

Create the IDE project

We need to create an IDE project for the program. But first, we need to create an IDE workspace. This workspace can also be used for other experimental projects created for the book. Start the IDE. A workspace prompt will appear, named **STM32CubIDE Launcher**. Or, the prompt can be opened using **File : Switch Workspace : Other**. Specify the path for the workspace to be created. The path used here is: `C:\projects\packtBookRTOS2E_experiments`.

In the IDE, login to ST by clicking on **Help : STM32Cube Updates : Connection to myST**.

Next, we'll create a project by clicking on **File : New : STM32 project**. An IDE GUI will appear, named **Target Selection**. In it, we'll specify our dev-board. Click on the **Board Selector** tab. At the drop-down box labeled **Commercial Part Number**, select the NUCLEO-F767ZI. Under **Boards List**, click on the dev-board. Then, click on **Next**. (It might take several minutes for downloads to complete here.)

In the GUI named **Setup STM32 project**, enter the project name. We'll use `Chapter_05_experiment`. Use the default settings shown below, and click on **Finish**.

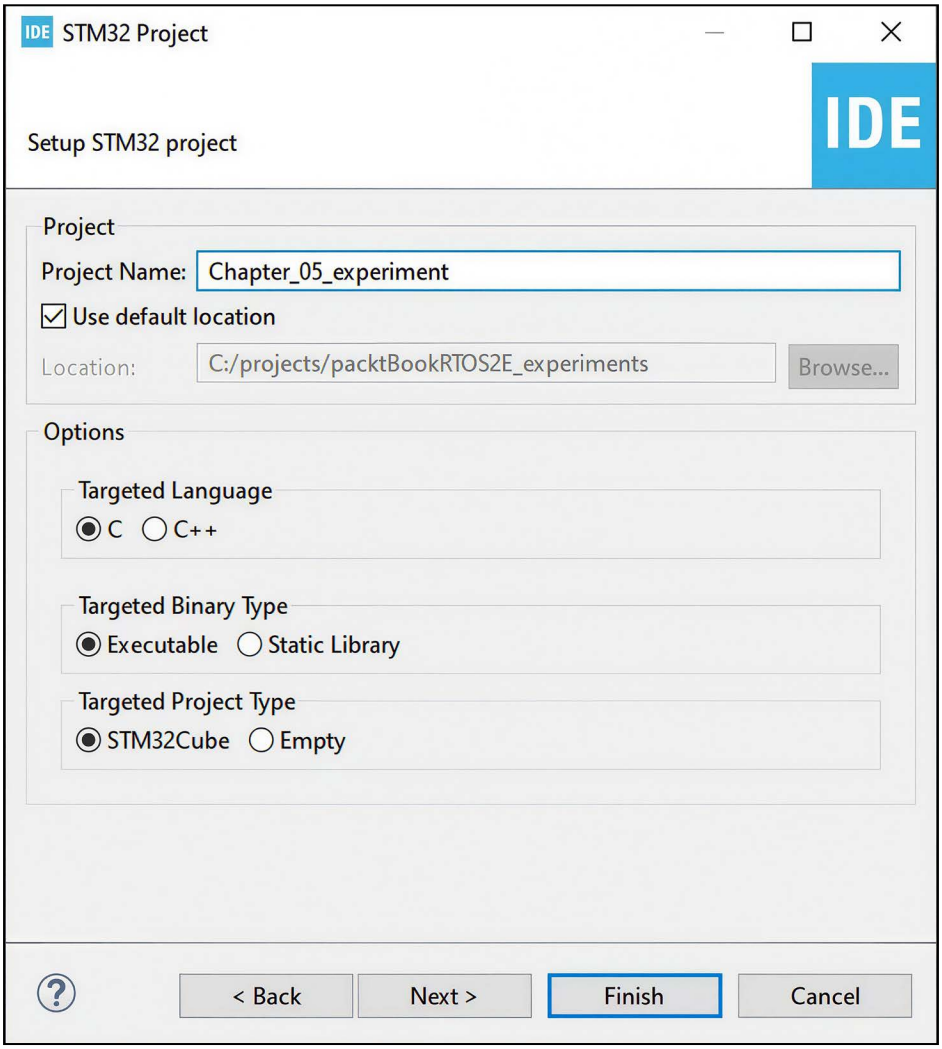


Figure 5. 2 : Specifying the project name

When prompted to **Initialize all peripherals with their default Mode?**, click on **No**. This will prevent the code-generator from creating extra code that we won't need. For any additional prompts, accept the default. (If problems are encountered, it can be helpful to take screenshots of the pop-up windows, and follow the recommendations for fixes.)

Project Explorer should look something like *Figure 5. 3*.

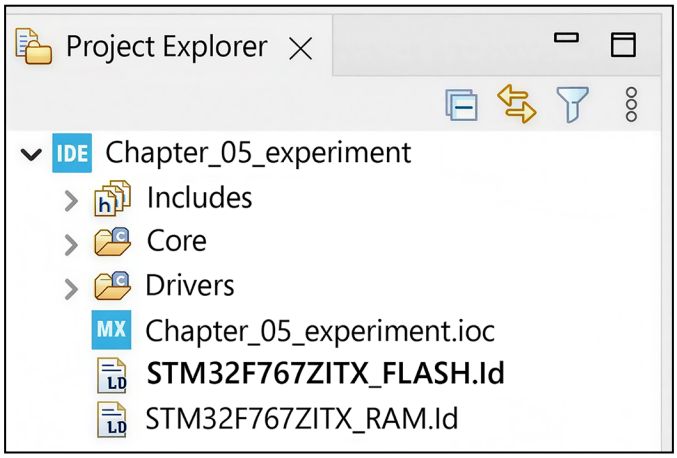


Figure 5. 3 : The created project

Configure the hardware

Next, the IDE's hardware-configuration GUI is displayed. The hardware configuration is stored in the file `<project_name>.ioc` (i.e., `Chapter_05_experiment.ioc`). Note there is a tab for the file, just above the hardware-configuration GUI.

Within the GUI, click on the tab **Clock Configuration**. The MCU's clock system is displayed. We'll start with the default clock configuration, and set the CPU to run at the maximum speed of 216 MHz. The clock system is complex, but it is not as complicated as it looks. Explaining the clock system itself is beyond the scope of this book. For those interested, the book *Mastering STM32* has a chapter on the clock system, and the chapter provides enough information to start working with the clock system on one's own.

To set the CPU clock, find the text-box labelled **HCLK (MHz)** in the clock configuration. The text box has a blue border, and in the figure below, it is pointed to by the red arrow. Set the text box to 216 and press `Enter`. At the prompt to reconfigure the clocks, click **Ok**.

The IDE will configure the clocks as shown in the two figures below. The second figure is a continuation of the first figure. (The first figure's right edge aligns with the second figure's left edge.) What is shown in the IDE's tab **Clock Configuration** should be the same as these figures.

Using these figures, the resultant clock-rates can be explained. There are several clocks on the dev-board and MCU. By default, the *High-Speed Internal Clock* is used as the clock source. It's on the MCU, and it's an RC oscillator. In the first figure, this clock is shown on the far left, and it is labelled **HSI RC**. It runs at 16 MHz.

In the two figures, the yellow-line shows the clock path from the **HSI RC** to the CPU (labelled **core**). So, the CPU will run at 216 MHz. The green-line shows the clock path to the SysTick timer (labelled **Cortex System timer**). The SysTick timer also runs at 216 MHz.

The HSI RC clock is used for convenience. In our experience, onboard RC clocks tend to be much less accurate than an external crystal, which would be used instead.

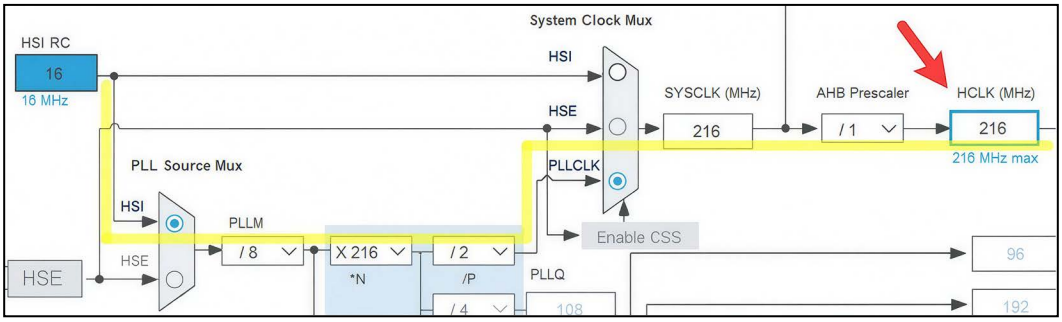


Figure 5. 4 : Clock configuration, part 1

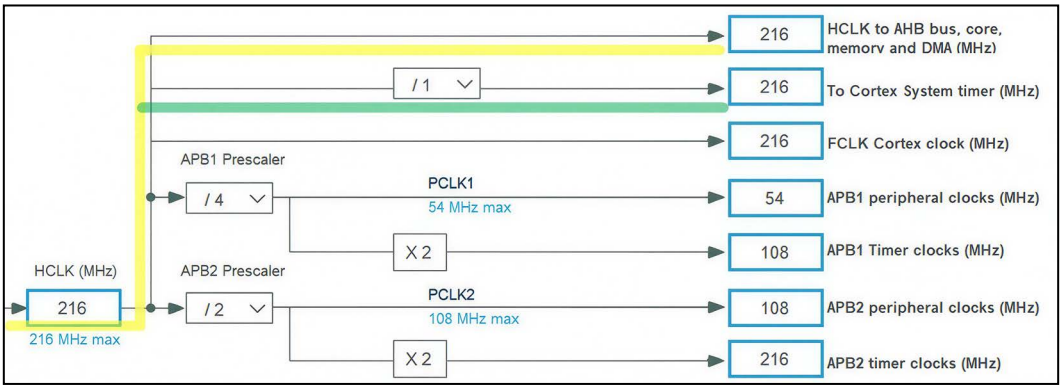


Figure 5. 5 : Clock configuration, part 2

The example-program uses the MCU's RNG. It is an MCU peripheral. To use it, it needs to be enabled, and that can be done with the hardware-configuration GUI. Click on the tab **Pinout & Configuration**, then the tab **A -> Z**. In the drop-down box, select **RNG**. The configuration dialog is then presented, labelled **RNG Mode and Configuration**. Click on the checkbox, **Activated**.

Activating the RNG causes conflicts with the default clock-settings that we're using. This is indicated on the **Clock Configuration** tab, by the red "x". However, the clock settings are easily fixed. Click on the **Clock Configuration** tab. There will be a prompt, **Do you want to run automatic clock issues solver?**; click on **No**. We'll fix the clock settings manually.

The RNG's clock requirements are specified in the MCU reference manual. It has a chapter on the RNG peripheral. We'll set the RNG clock to 48 MHz, as required. The figure below shows the RNG clock. It's on the right side, with a blue border. Set the **PLLQ clock-divisor** to 9. This sets this **PLLQ clock** to 48 MHz. The **PLLQ clock** is configured to be the RNG clock, as indicated by the red circles.

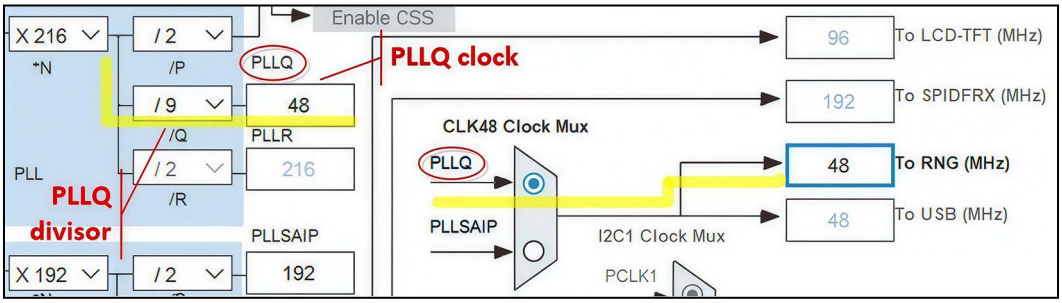


Figure 5. 6 : RNG clock configuration

The following equation and *Figure 5. 7* illustrate the calculations involved in setting the `PLLQ-clock`'s rate to 48 MHz. The clock division-factors (`PLLM` and `PLLQ`) and multiplication-factor (`PLLN`) are described in the MCU reference manual.

```
PLLQ_clock_rate = ( ( HSI_clock_rate / PLLM ) * PLLN ) / PLLQ_clock_
divisor
48 MHz = ( ( 16 MHz / 8 ) * 216 ) / 9
```

In the figure, we see the PLLQ-clock's rate is set to 48 MHz, as in the equation.

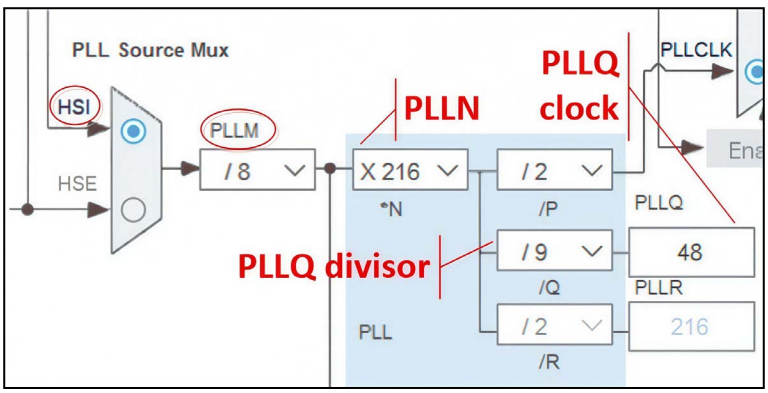


Figure 5. 7 : How the PLLQ clock-rate is set

Finally, save the *.ioc file, using `Ctrl+S`. There will be a prompt, **Do you want to generate code?**. Click **Yes**. For any additional prompts, select the default. The generated code includes `main.c`, and it will be shown in a tab. The *.ioc tab can be closed.

Create the user-code

Now that the project's generated-code has been created, it can be viewed via Project Explorer, as shown below.

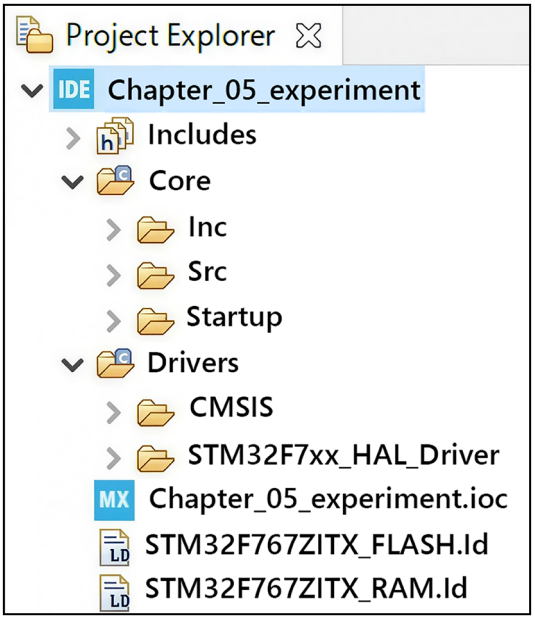


Figure 5. 8 : Project Explorer, with generated code

In the figure, the folders are:

- `Includes`: shows all include-file folders used by GCC
- `Core`: generated code for the MCU configuration
- `Core/Src/main.c`: configures and initializes the MCU
- `Drivers`: CMSIS and HAL library-files

For the IDE, code-generation is described in the ST manual *STM32CubeMX for STM32 configuration and initialization C code generation*. STM32CubeMX is the tool that performs code-generation. It is a standalone tool, and it has also been made part of STM32CubeIDE.

Next, the user-code can be added to the project's program files. The timed super-loop's code was shown earlier in this chapter, and it is in the example-program's files `main.c` and `stm32f7xx_it.c`. Also, the header-file `main.h` has a definition of the variable `timerDone`, which is used by the ISR. The easiest way to implement the example-program here is to copy all of that code from the example-program's project and into the present project, in the respective files.

It is safest to copy in just the files' user-code and not copy the whole files. (This can be done using a file-compare tool, with the ability to copy selected text, e.g., Beyond Compare.) There's a possibility that the code generated by your STM32CubeIDE is somewhat different than the code generated for the book's Chapter_5 example-program. This can be caused by using different versions of STM32CubeIDE, and those differences should be benign.

In general, when adding or deleting project files and folders, it can be prudent to use the IDE's file-utilities, such as `New`, `Copy` and `Delete`. This way, the IDE can also update the build scripts, if necessary. One way to use those utilities is by right-clicking in the IDE's Project Explorer.

The chapter's example-program also includes the user-written function `delay_us`. It's an in-line function defined in `Core/Inc/systick_timer_us_delay.h`. To add this header file to the present project, right-click on `Core/Inc`, and click on **New** : **Header file**. Enter the file name, `systick_timer_us_delay.h`, and click on **Finish**. Copy the file contents from the example-program project, into this new header file.

Install and configure the debugger software

To use the debugger, we need to do two things: 1) an Ozone project-file needs to be created, and 2) the SystemView *target source-code* needs to be added to the IDE project. This target source-code is the SystemView code that is called by the super-loop program, and when built, it is part of the firmware executable.

Create an Ozone project file (*.jdebug) as described in chapter 3. The full path to the *.elf file will be needed. It will be in the project at `Debug\Chapter_05_experiment.elf`.

Install the SystemView target source-code

Adding the target source-code to the IDE-project involves: a) downloading the source-code, b) adding the source-code to the project, and c) setting-up the project's build configuration for the source-code.

How to install the target source-code on a target (e.g., a dev-board) is described in the SystemView user guide. To simplify the installation, instructions are provided here for our dev-board. These instructions have been tested with SystemView v3.60d and Target Sources v3.60d.

Download the SystemView "Target Sources" from `https://www.segger.com/downloads/systemview`. Select a version that is the same as, or less than, the version of your SystemView app (e.g., v3.60d). Unzip the downloaded file. There should be a source-code directory with a name that is something like `SystemView_Src_V360d`.

In the IDE, we'll create a folder for the SystemView source-code. Open the project and click on **File : New : Folder**, then enter the folder-name "`Middleware`".

Next, we'll import the source code, using the IDE's import wizard. In the Project Explorer, right-click on the folder `Middleware`, and select **Import**. In the import wizard, select **General : File System : Next**.

In the **Import** dialog, for the text-box **From directory**, provide the path to the SystemView source-code directory, and press `Enter`. Under **From directory**, click on the check-box for the source-code directory (e.g., `SystemView_Src_V360d`). For the text-box **Into folder**, specify the path to the folder `Middleware` (i.e., `Chapter_05_experiment/Middleware`).

Click on the check-box **Create top-level folder**, then click on **Finish**. The SystemView source-code will then be imported, under the `Middleware` directory, i.e., at `Middleware/SystemView_Src_V360d`. Use the IDE to rename the source-code directory to `SystemView`. Project Explorer should now show `Middleware/SystemView`.

Next, the SystemView source-code needs to be configured for the OS being used. SystemView provides configuration files for various operating systems. Configuration files are also provided for MCUs with no operating system. We'll use the configuration file that is for systems with: a) no operating system, and b) a Cortex-M processor. That configuration file is at: `Middleware/SystemView/Sample/NoOS/Config/Cortex-M/SEGGER_SYSVIEW_Config_NoOS.c`

Use the IDE to copy that file to the directory: `Middleware/SystemView/Config/`.

Finally, use the IDE to delete the directory with the OS configuration-files: `Middleware/SystemView/Sample`. (If this directory is not deleted, the build system will try to compile the files in there.)

The `Middleware` directory should look like this:

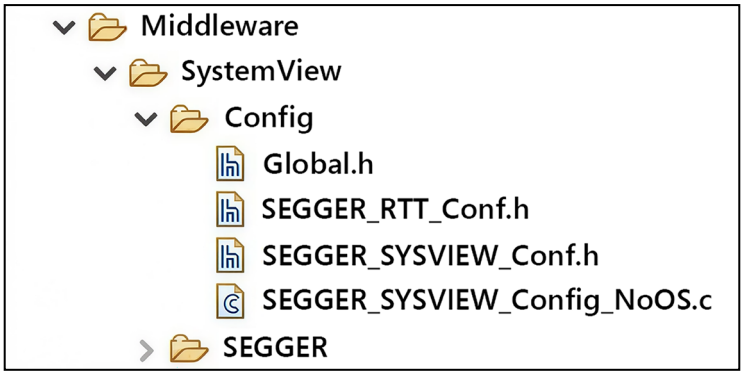


Figure 5. 9 : SystemView target source-code, installed

Add the build-configuration settings

Next, the IDE project's build-configuration needs to be set-up for the SystemView code that was just imported. The project's build-configuration is part of the project's Properties. By default, the project has two build-configurations: `Debug` and `Release`. We'll just set-up the `Debug` build-configuration. Make it the Active build-configuration, by clicking on **Project : Build Configurations : Set Active : Debug**.

First, we'll configure the project's compiler-settings. Open the project's Properties by clicking on **Project : Properties**. In the leftmost frame click on **C/C++ Build : Settings**. Under the tab **Tool Settings**, click on **MCU GCC Compiler : Include paths**. In the rightmost frame, add the paths for the SystemView include-files, as shown below. The include-paths are `../Middleware/SystemView/Config` and `../Middleware/SystemView/SEGGER`.

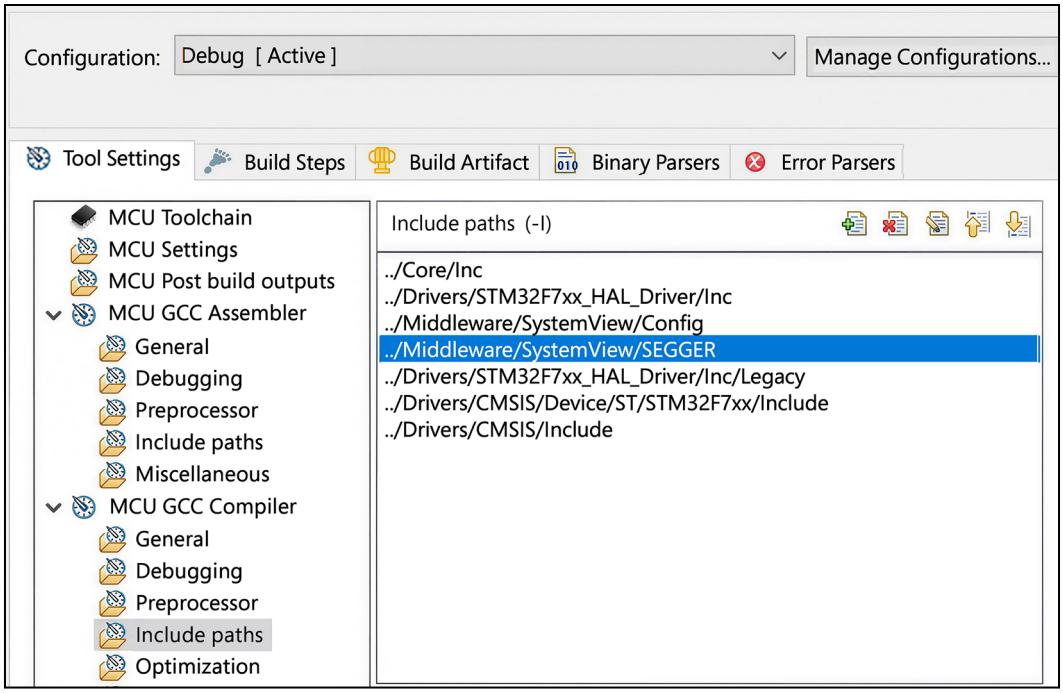


Figure 5. 10 : GCC compiler include-paths

Those two include-paths also need to be added for the assembler. Click on **MCU GCC Assembler : Include paths**, and add the paths.

In the leftmost frame, click on **C/C++ General : Paths and Symbols**. Then click on the tab **Includes**. Under **Languages**, click on **GNU C**. It should show the SystemView directories that you just entered.

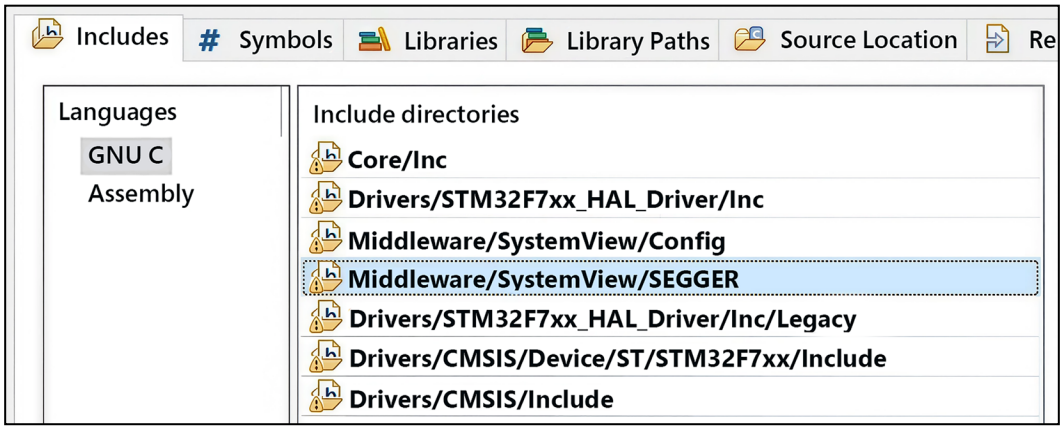


Figure 5. 11 : Verifying the include-paths

Under the tab **Source Location**, add the folder for the SystemView source-code that is to be compiled: `/Chapter_05_experiment/Middleware/SystemView`.

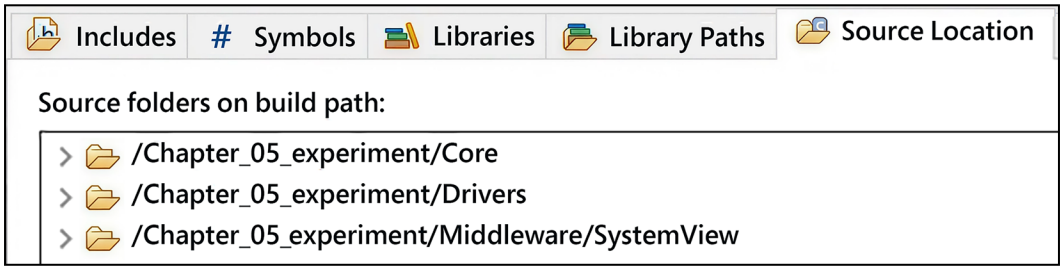


Figure 5. 12 : Configuring the source location

Click on **Apply and close** to close the Properties. Save the project, if needed. Next, we can build and run the super-loop.

Build and run the firmware

The super-loop program can be built and run in the same manner as the example-program, which was described earlier in the present chapter.

When running the super-loop program, recall that the super-loop initialization will spin until the SystemView app is started on the development-computer, in Record mode.

Using an oscilloscope to measure timing

An oscilloscope can be used to measure the super-loop's timing. How to do that is introduced here. This section is intended for those unfamiliar with oscilloscope-use. Seeing how to get started with a new technology can make it much more approachable. Just beware that incorrectly connecting wires to the dev-board can potentially damage it.

The oscilloscope used is a PicoScope 2204A. They are relatively inexpensive scopes (currently under $200), and they can be useful for getting started with embedded-systems and electronics. However, an oscilloscope is not needed for this book, and it's not used in subsequent chapters.

The super-loop code sets the green user-LED on and off, just before and after the delay function, respectively. We'll connect the oscilloscope to the LED, and use it to measure the super-loop's delay, iteration period, and processor utilization.

Connect the oscilloscope to the LED

The figure below shows how the oscilloscope can be connected to the LED on the dev-board. The LED is electronically connected to a connector-pin on the dev-board. That connector-pin is `PB0`. The oscilloscope probe is connected to `PB0` via the red jumper wire. Here, a small breadboard was used to connect the probe to the jumper wire. The breadboard is just used to prevent the probe and jumper wire from wiggling around.

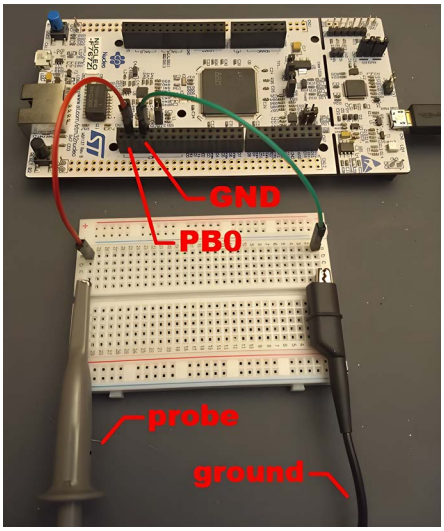


Figure 5. 13 : Connecting an oscilloscope to the dev-board's LED

To connect the oscilloscope to the dev-board, we need to figure out what dev-board connector-pin is connected to the LED. First, we'll figure out what GPIO pin is connected to the LED. Then we'll figure out what connector-pin is connected to the GPIO pin.

In the dev-board's user manual, the green user-LED is named `LD1`. The user manual also specifies that the LED is connected to one of the MCU's GPIO pins. In particular, it is pin `PB0` (GPIO port B, pin 0). Explaining GPIOs is beyond the scope of the present book. The book *Mastering STM32* has a chapter on GPIOs, and it provides a basic working knowledge of them.

We can confirm the GPIO pin-assignment by examining the super-loop's initialization-code. In `main.c`, the function `MX_GPIO_Init` is generated by the IDE, and that function contains the GPIO initialization-code. Perusing the code, we see an `LD1_Pin` being initialized, and it is in GPIO port B, and it's pin 0. (In the super-loop code, the function `HAL_GPIO_WritePin` is used to turn the LED on and off. When writing that code, the dev-board's user manual would be consulted in a similar manner, to determine what GPIO pin to use.)

Next we need to figure out what dev-board connector-pin is connected to the GPIO pin. This is specified in the dev-board's user manual. The dev-board has many connector-pins. The connector-pin `PB0` is in the ST Zio connector `CN10`. In the figure above, `CN10` is the black plastic block that the jumper wires connect to. The user manual specifies the `CN10` pin layout, and it's shown below. It specifies that `PB0` is at connection-pin 31, and `GND` is at connection-pin 27.

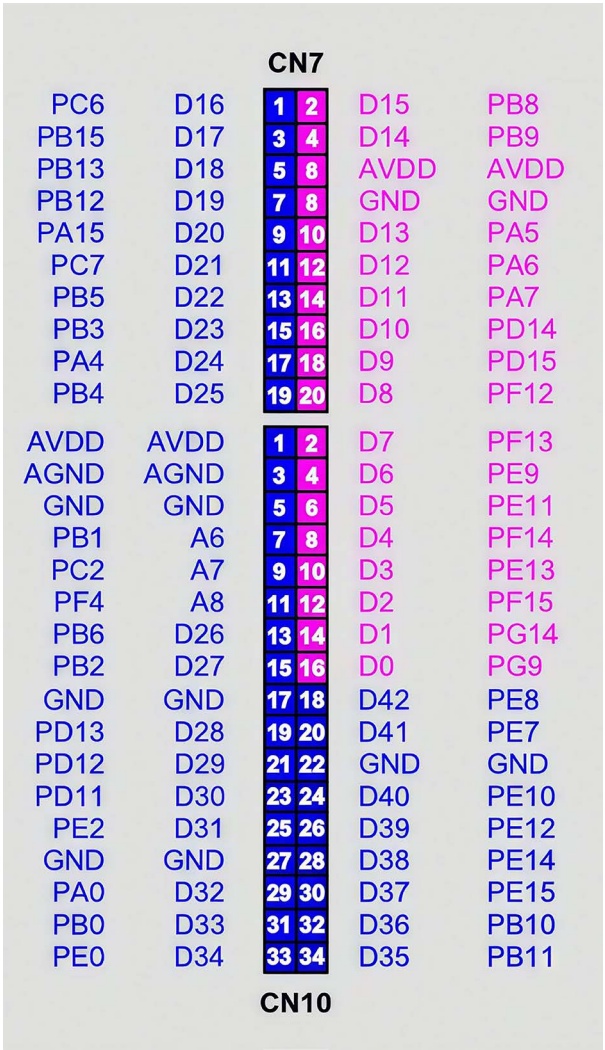


Figure 5. 14 : Pin lay-out for connector CN10

In the figure with the dev-board (*Figure 5. 13*), the black oscilloscope lead is connected to the green jumper wire, which is connected to ground on the dev-board, at connector-pin `GND` (pin 27). The dev-board's user manual specifies which connector-pins are ground (`GND`).

Run the oscilloscope

A screenshot of the running oscilloscope is shown below. A trigger is used to begin capture. The trigger condition is a 3V threshold, with a rising signal. This occurs when the LED is turned-on. The repeat trigger-mode is used. So, as possible, the scope displays data whenever the trigger condition is met (i.e., whenever the LED is turned-on). The capture indicates the timing requirements are being met. The measured iteration-period is around 977 microseconds, which is sufficiently close to 1 ms. The range for the LED's period (pulse width) is sufficiently close to 200 to 600 microseconds. Recall that the super-loop's processor utilization was defined as the iteration's non-sleep time divided by the iteration-period. The measurements here are consistent with the processor-utilization calculations shown earlier, i.e., anywhere from 20% to 60%.

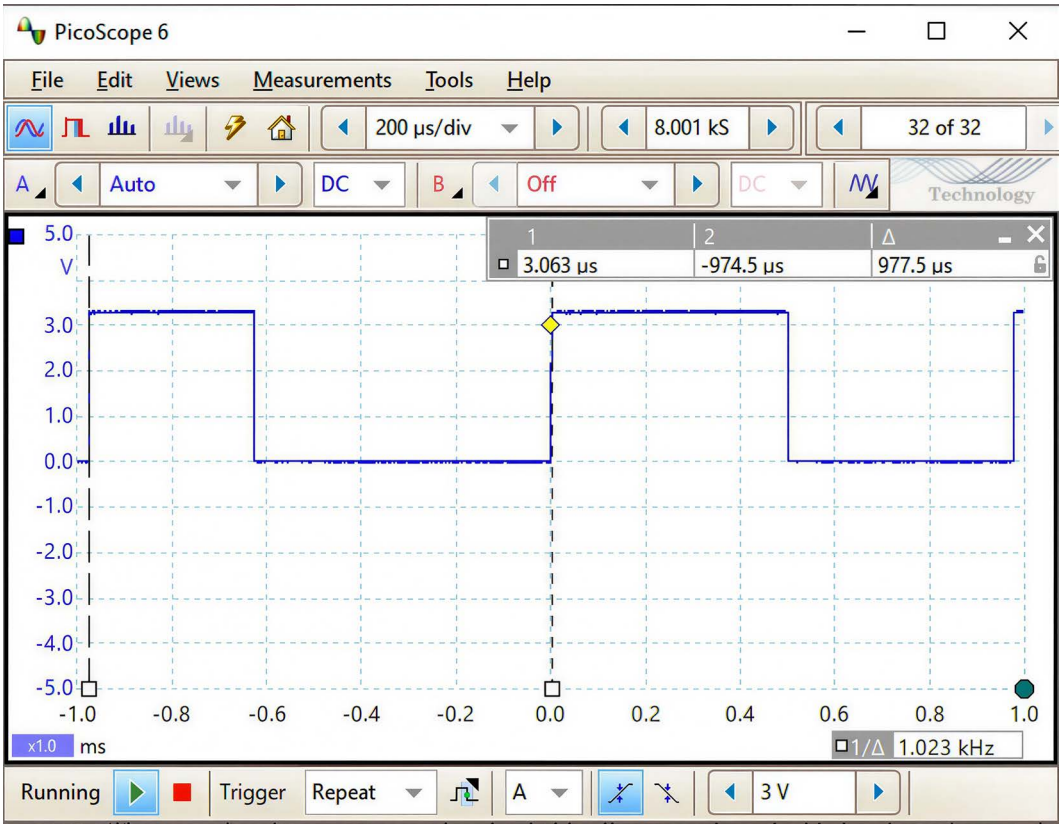


Figure 5. 15 : Oscilloscope capture

The present section was primarily intended for those new to embedded systems, to provide context for understanding how the book's example-programs are created. Fully explaining some topics was out-of-scope. However, references were provided to show those topics are approachable and can readily be learned.

Summary

The example-program showed how the timed super-loop can be implemented on our dev-board and MCU. And, although the timed super-loop is conceptually simple, the implementation is relatively complex, due to a number of factors, such as a) the ISR running concurrently with `main()`, b) how the sleep function works relative to interrupts, c) the need for memory-barrier instructions, d) the potential for timing problems from the debugger, and e) the challenges related to determining how system-APIs work.

For those new to embedded-systems development, or new to the MCU and IDE, the present chapter provides a basic understanding of how the book's example-programs were created, and how to further research the code, such as the system-API calls.

In the next chapter, we will introduce RTOS tasks and show their benefits relative to super-loop programming.

Part 2

The FreeRTOS Scheduler, Tasks, and Primary APIs

In Part 2 of the book, you'll learn about the RTOS named FreeRTOS and its primary components. With FreeRTOS, multiple programs can share a single processor. The programs run concurrently and give the impression of running in parallel. A task is created to run each program, and the FreeRTOS scheduler determines which task is currently running. Tasks often need to synchronize their processing, and FreeRTOS provides features for that, including semaphores and mutexes. In addition, tasks often need to send each other data, and FreeRTOS's queues provide that capability. By the end of Part 2, you'll have a good understanding of these FreeRTOS components. Example programs are provided, and with them, you'll have the opportunity to get hands-on experience with the FreeRTOS components.

Part 2 includes the following chapters:

- *Chapter 6*, *Understanding RTOS Tasks*
- *Chapter 7*, *Running the FreeRTOS Scheduler*
- *Chapter 8*, *Protecting Data and Synchronizing Tasks*
- *Chapter 9*, *Intertask Communication*

6

Understanding RTOS Tasks

In the last two chapters we saw how bare-metal firmware applications make use of the super-loop and the timed super-loop to perform their duties in real time. We also saw how a function can be broken up by using a state machine. This allows the super-loop to perform the function's duties without preventing other functions from running in a timely manner. In this chapter, we introduce RTOS tasks and show how they make it possible for functions to perform their duties while meeting their deadlines.

The present chapter consists of three parts. The first part provides a conceptual view of the FreeRTOS scheduler and tasks. The second part describes how FreeRTOS was chosen for the book's RTOS. The third part presents a FreeRTOS example-program, how to run it, and how to use SystemView to analyze task execution. FreeRTOS installation and configuration are also presented.

The technical requirements for this chapter are:

- The dev-board with J-Link installed, from chapter 3.
- The software tools that were installed in chapter 3: STM32CubeIDE, Ozone, and SystemView.
- The system documentation listed in *Appendix B*, up to the present chapter.

The chapter's major sections are:

- The FreeRTOS scheduler and tasks: a conceptual view
    - The need for something more than a super-loop
    - Comparing RTOS tasks to super-loop functions
    - Changing super-loop functions into RTOS tasks
- Choosing FreeRTOS as our RTOS

- Hands-on with FreeRTOS's scheduler and tasks
    - Understanding and working with the example-program
    - Installing and configuring FreeRTOS

The FreeRTOS scheduler and tasks: a conceptual view

Below, the first section introduces the limitations of super-loops and the need for an RTOS. The second section shows how an RTOS extends a super-loop's basic structures. The third section describes RTOS schedulers and tasks, and how they work.

The need for something more than a super-loop

The super-loop is a real-time system that can handle events and process interrupts, while still meeting timing deadlines. It seems like we achieved a lot with a super-loop, so why even consider something else like an RTOS? Well, it is entirely possible you don't need to consider anything else! If the system has a limited number of responsibilities and none of them are especially complicated or time-consuming, then there may be no need for anything more sophisticated than a super-loop.

However, an RTOS has an advantage over a super-loop when a system needs to handle multiple events and meet multiple deadlines. RTOSes ease the management of multiple tasks that have different time constraints. An RTOS can schedule tasks to execute at different time intervals and with different priorities, in order to meet hard deadlines. And, while doing that, an RTOS can also execute tasks that have a soft deadline, like a **User Interface (UI)**.

Imagine a system that is responsible for generating a UI, running complex time-consuming algorithms, and dealing with complex communication stacks. It is very likely that these tasks will take a non-trivial amount of time. If a glitzy eye-catching UI with lots of animation starts to stutter a little bit because the MCU is dealing with collecting data from a critical sensor, it is no big deal. The animation can be dialed back or eliminated, and the important part of the real-time system can be left intact. In contrast, consider what might happen if the animation was made to look perfectly good, but at the cost of missed data from the critical sensor.

There are all sorts of different ways in which this problem plays out every-day in our industry. Sometimes, if the system was designed well enough, the missing data will be detected and flagged (but it can't be recovered: it is gone forever). If the design team is really lucky, the system may even fail in this way during in-house testing. However, in many cases, the missed sensor data will go completely unnoticed until somebody notices one of the readings seems to be a little bit off ... sometimes.

If everyone is lucky, the bug report for the sketchy reading might include a hint that it only seems to happen when someone is at the front panel (playing with those fancy animations). This would at least give the poor firmware engineer assigned to debug the issue a hint – but we're often not that lucky.

These are the types of systems where an RTOS is needed. Guaranteeing that the most time-critical tasks are always running when necessary is a strong point of preemptive schedulers. Scheduling lower-priority tasks to run whenever spare time is available is another strong point. In these types of setups, the critical sensor-readings could be pushed into their own task and assigned a high priority – effectively interrupting anything else in the system (except ISRs) when it is time to deal with the sensor. That complex communication stack could be assigned a lower priority than the critical sensor. Finally, the glitzy UI with the fancy animations gets the left-over processor cycles. It is free to perform as many sliding alpha-blending animations as it wants, but only when the processor doesn't have anything else better to do.

Another advantage of using an RTOS is that the RTOS provides all of the mechanisms for managing tasks whereas the super-loop programmer needs to create such mechanisms from scratch. A mature RTOS, like FreeRTOS, is actively developed to improve the overall quality of the code and to fix defects discovered by users. This maturity provides an increased confidence in the task-management portions of the code.

Another often-touted advantage of an RTOS is that it eases **portability**. Portability is the term used to describe how easy or difficult it is to move code from one MCU to another. When writing code that executes on top of an RTOS, much of the code is portable no matter which MCU is being used. In a sense, porting that code is handled by the RTOS, freeing up the programmer to concentrate on the unique application code being written. An RTOS is typically ported to a specific MCU's CPU, and a specific compiler. Portions of the kernel are written in assembly language, in order to take advantage of the CPU features and instructions. This often results in very efficient code, which is important in order to minimize the processor cycles used by the task-management and scheduling features. Also, an RTOS can shield the programmer from having to deal with errata published by the CPU vendor, and from complex coding requirements such as memory barriers. However, MCUs have differing peripheral registers and features. Since real-time software tends to interact heavily with these peripherals, portability of application code is often difficult to achieve in real-time systems, relative to general computing systems.

One disadvantage of using an RTOS is that the programmer must learn how to use the RTOS (maybe that is why you are reading this book!). A typical RTOS is complex and provides functions and features for various applications. It will take some effort to get up the learning curve – reading books, studying code, developing prototypes, searching the internet, asking questions in forums, etc. But in the end, the skills you acquire will allow you to build applications on the RTOS you learn, and the skills will give you a good understanding of how RTOSes work in general, so that you can quickly adapt the skills to other RTOSes.

Now that we've made the case for using an RTOS, let's compare RTOS tasks to super-loops.

Comparing RTOS tasks to super-loop functions

So far, we've only mentioned RTOS tasks very casually, but what is a task, really? An easy way to think about an RTOS task is that it is *just another main loop*. In a preemptive RTOS, there are four main differences between tasks and super-loops:

- Each task receives its own private stack. Unlike a super-loop in `main`, which was sharing the system stack, tasks receive their own stack that no other task in the system will use. This allows each task to have its own call stack without interfering with other tasks.
- Each task has a priority assigned to it. This priority allows the **scheduler** to make decisions on which task should be running (the goal is to make sure the highest priority task in the system is always doing useful work). All of the functions inside of a super-loop, however, run at the same priority and always execute each time through the loop.
- Each task may be interrupted by other tasks. This is unlike the super-loop, whose functions run in a predetermined serial order.
- Tasks in an RTOS may exceed the scheduler tick period. This allows for tasks with longer execution times to be partitioned by the RTOS. In contrast, with a super-loop, functions with very long execution times must be partitioned by the programmer, e.g., by using a state machine.



The FreeRTOS scheduler will be discussed in more detail in *Chapter 7*, *Running the FreeRTOS* Scheduler.

Given these three features, each task may be programmed as if it is the only thing the processor has to do. Do you have a single flag you'd like to watch AND some calculations for flashy animations to churn through? No problem: simply program the task and assign it a reasonable priority, relative to the rest of the system's functionality.

The preemptive scheduler will always ensure that the most important task is executing when it has work to do. When a higher priority task no longer has useful work to perform and it is waiting on something else in the system, a lower priority task will be switched into context and allowed to run. This was not possible with the super-loop, as one function needed to finish before the next could run.

An RTOS can be thought of as extending the super-loop's capabilities. In a super-loop, the loop is analogous to the RTOS scheduler. Also, the super-loop's functions are analogous to RTOS tasks, as each function in the super-loop performs some desired "task". The super-loop schedules each function to run every loop period. The super-loop's functions all operate at the same priority, and the functions are performed serially. The difference with an RTOS is that the RTOS tasks can be run concurrently, and they are assigned different priorities, for determining which RTOS task is run next.

Changing super-loop functions into RTOS tasks

In the previous chapter, we looked at a super-loop that was looping through three functions. Now, for a very simple RTOS example, let's move each one of the three functions into its own task. We'll use those three simple tasks to examine the following:

- **Conceptual task-programming model**: How the three tasks can be described theoretically
- **Actual round-robin scheduling**: What the tasks look like when executed using a round-robin scheduling algorithm
- **Actual preemptive scheduling**: What the tasks look like when executed using preemptive scheduling



In real-world programs, there is almost never a single function per task; we're only using this as an analog to the overly simplistic super-loop from earlier.

A conceptual task-programming model

Here's some pseudo-code that uses a super-loop to execute three functions. The same three functions are also included in a task-based system – each RTOS task (on the right) contains the same functionality as the functions from the super-loop on the left. This pseudo-code will be used moving forward, as we discuss the differences in how the code is executed when using a super-loop versus using tasks and a scheduler:

Super Loop

```
func1()
{
    // functionality 1
}
func2()
{
    // functionality 2
}
func3()
{
    // functionality 3
}
main()
{
    While(1)
    {
        func1();
        func2();
        func3();
    }
}
```

RTOS Tasks

```
Task1()
{
    while(1)
    {
        // functionality of Task 1
    }
}
Task2()
{
    while(1)
    {
        // functionality of Task 2
    }
}
Task3()
{
    while(1)
    {
        // functionality of Task 3
    }
}
Main()
{
    createTask(&Task1);
    createTask(&Task2);
    createTask(&Task3);
    StartScheduler();// never return
}
```

Table 5.1: Comparing the functions of a super-loop to RTOS tasks

One of the immediate differences you might notice between the super-loop implementation and the RTOS implementation is the number of infinite `while` loops. There is only a single infinite `while` loop (in `main()`) for the super-loop implementation, but each task has its own infinite `while` loop.

In the super-loop, the three functions being executed are each run to completion before the next function is called, and then the cycle continues onto the next iteration (illustrated by the following diagram):

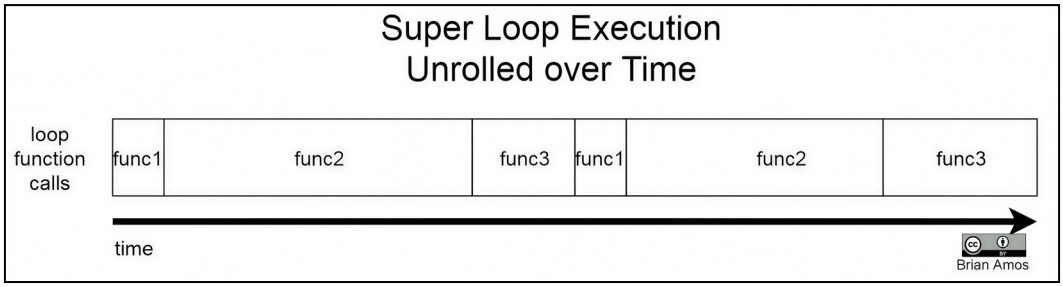


Figure 6.1 : Super-Loop Execution Unrolled over Time

In the RTOS implementation, each task is essentially its own little infinite `while` loop. Whereas functions in the super-loop were always sequentially called one after the other (orchestrated by the logic in the super-loop), tasks can simply be thought of as all executing in parallel after the scheduler has been started. Here's a diagram of an RTOS executing three tasks:

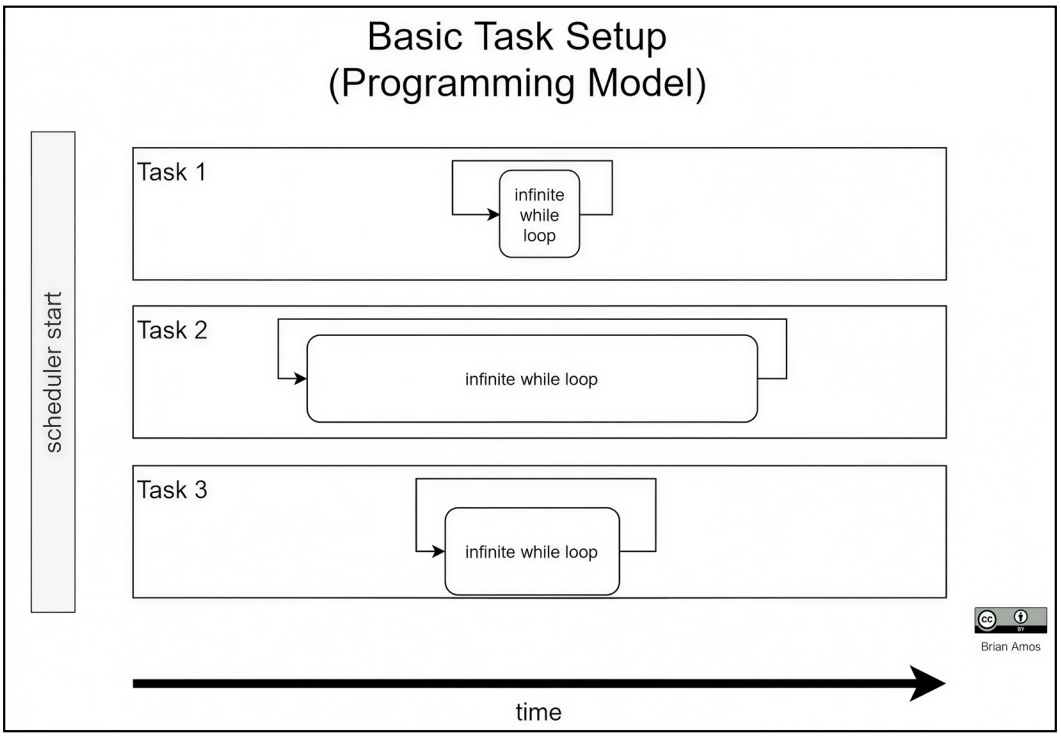


Figure 6.2 : Basic Task Setup (Programming Model)

In the diagram, you'll notice that the size of each `while` loop (i.e., the iteration period) is not the same. This is one of the many benefits of using a scheduler that is executing the tasks in *parallel* versus a super-loop – the programmer doesn't need to be immediately concerned about the loop with the longest iteration-period impacting loops with shorter iteration-periods. The diagram depicts `Task 2` as having a much longer iteration-period than `Task 1`. In a super-loop system, the functionality in `func1` would execute less frequently than it would with the tasks in parallel execution (since the super-loop would need to execute `func1`, then `func2`, and then `func3`). In a task-based programming model, with parallel execution, this isn't the case – the loop of each task can be thought of as being isolated from the other tasks in the system – and they all run in parallel.

Strictly speaking, the three tasks run in **parallel** when they all run at the same time. So, three processors would be needed. On a system with a single processor, three tasks can run **concurrently**, but not in parallel. With **concurrent execution**, the three tasks take turns using the single processor. However, in practice, concurrent execution often outwardly appears like parallel execution.

This task-isolation and perceived parallel execution are some of the benefits of using an RTOS; it alleviates some of the complexity for the programmer. So – that's the easiest way of conceptualizing tasks – they're simply independent infinite `while` loops that all execute in parallel . . . in theory. In reality, things aren't quite this simple. In the next two sections, we'll take a glimpse into what goes on behind the scenes to make it *seem* like tasks are executing in parallel.

Round-robin scheduling

One of the easiest ways to conceptualize actual task execution is with round-robin scheduling. In round-robin scheduling, each task gets a small slice of time to use the processor, which is controlled by the scheduler. As long as the task has work to perform, it will execute. As far as the task is concerned, it has the processor entirely to itself:

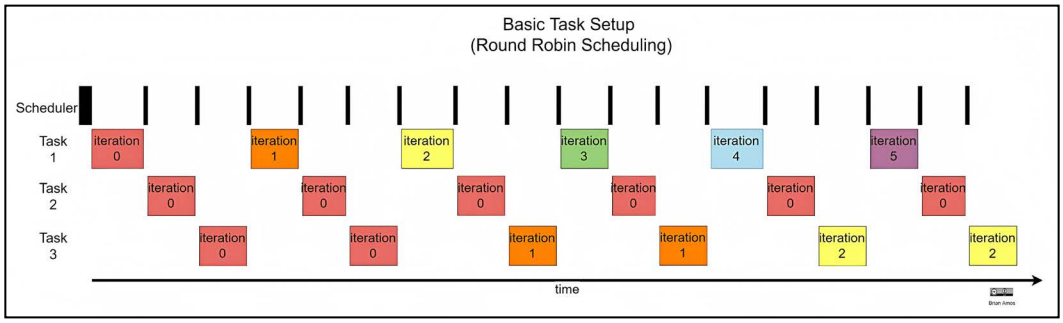


Figure 6.3 : Basic Task Setup (Round Robin Scheduling)

This is the same three tasks that were shown previously. However, it is a more detailed model, and each iteration of the tasks' loops are enumerated over time. Because the round-robin scheduler assigns equal time slices to each task, the shortest task (`Task 1`) has executed six iterations of its loop, whereas the task with the slowest loop (`Task 2`) has only made it through the first iteration. `Task 3` has executed three iterations of its loop.

An extremely important distinction between a super-loop versus a round-robin scheduler is that the functions in a super-loop need to wait for the other functions to finish. In contrast, functions in round-robin scheduling run concurrently, and each gets an equal share of the processor time. A limitation of this scheduling scheme is that a task with a shorter loop will execute more loop-iterations than a task with a longer loop.

All of this task management does come at a cost. Changing the running task is referred to as a **context switch**, and it's performed by the RTOS scheduler. For a context switch, the scheduler saves the current task's state, such its stack pointer, CPU registers, and program counter. And, the scheduler loads the next task's state, e.g., stack pointer, CPU registers, and program counter. Also, each time the scheduler runs, it performs additional duties, such as determining what task is to be run next. This means that there is a slight use of processor bandwidth every time the scheduler is invoked and every time it performs a context switch.

In this example, the tasks are not invoking the scheduler themselves. Instead, the scheduler is invoked via the CPU's SysTick timer. It is a count-down timer, and it counts down to 0 every milli-second. When the timer reaches zero, the SysTick ISR is run, and the ISR runs the scheduler. Also, when the timer reaches zero, it is reset to its starting value, and the countdown resumes. The moment when the SysTick timer reaches zero is referred to as the **system tick** (or just **tick**). The term **scheduler tick** refers to the periodic running of the scheduler, at the system tick.

The term **tick** is also used as a unit-of-measure for system time, and its duration is the SysTick timer's period, e.g., 10 ticks. A tick period of 1 milli-second is typical, and assumed here. In FreeRTOS, the tick-period is configurable, and a different tick-period can be used. This description of the SysTick timer applies to ARM Cortex-M7 CPUs, like the one on our dev-board. (More details can be found in *Chapter 7*, *Running the FreeRTOS Scheduler*).

A considerable amount of effort is put into making sure the scheduler kernel code is extremely efficient and uses the processor as little as possible. However, the fact remains that the scheduler will run at some point and consume CPU cycles. On most systems, the small amount of overhead is generally not noticed (or significant), but it can become an issue in some systems.

Preemptive-based scheduling

Preemptive scheduling provides a mechanism for ensuring that the system is always performing its most important task. Unlike round-robin scheduling, which gives an equal share of processing time to each task, a preemptive scheduling algorithm executes the highest priority task (interrupts are separate from tasks and generally have higher priority than tasks). The priority of each task is determined by the programmer, and task priorities need to be designed carefully. Whether the system will be able to meet its time constraints requires the skillful partitioning of code into tasks and ISRs, along with assigning proper priorities. While this may sound straightforward, there are pitfalls into which the programmer may fall, such as priority inversion, task starvation, etc.

To illustrate preemptive scheduling, let's take a look at the same three tasks that were previously presented with round-robin scheduling. Let's assume that each task requires more execution time than the scheduler tick time, so we can see how a preemptive scheduler handles scheduling and priority.

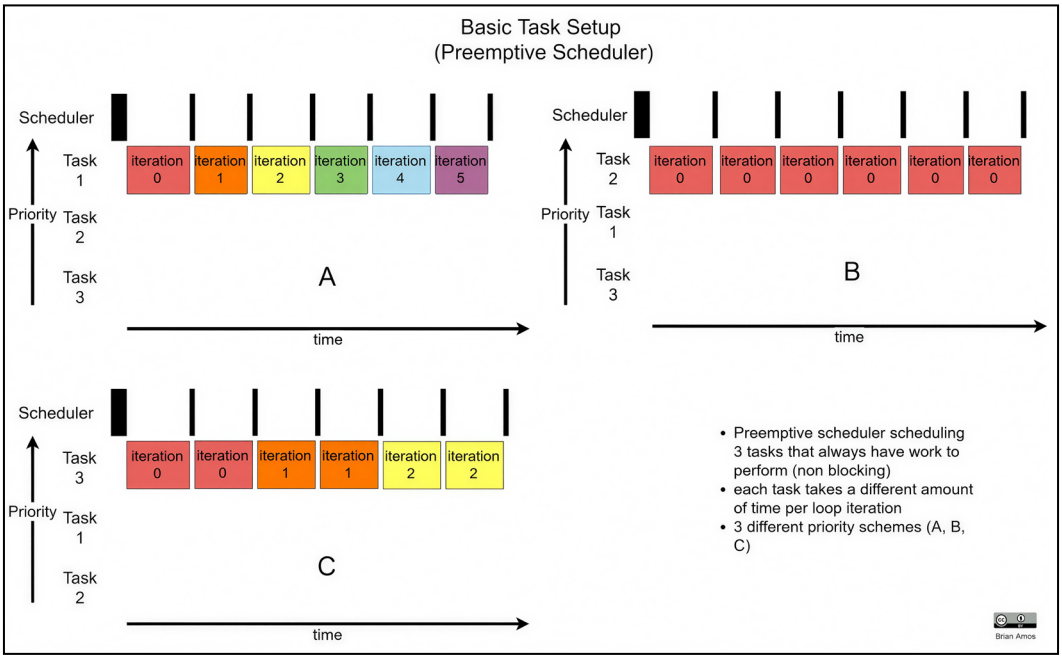


Figure 6.4 : Basic Task Setup (Preemptive Scheduler)

What happens when three different tasks are set up with three different sets of priorities (sets A, B, and C)?

- **A (top left)**: `Task 1` has the highest priority in the system – it gets *all* of the processor time! Regardless of how many iterations `Task 1` performs, if it is the highest priority task in the system and it has work to do (without waiting on anything else in the system), it will be given context and run.
- **B (top right)**: `Task 2` is the highest priority task in the system. Since it has more than enough work to do, and does not need to wait on anything else in the system, `Task 2` will be given context. Since `Task 2` is configured as the highest priority in the system, it will execute until it needs to wait on something else in the system.
- **C (bottom left)**: `Task 3` is configured as the highest priority task in the system. No other tasks run because they are lower priority.

Now, obviously, if you only have one processor, and you are designing a system that requires its three tasks to actually run in parallel, all of the time, a preemptive scheduler wouldn't be of much use. The above setup also would not be a great design for a real-time system since it would be completely overloaded (and ignoring two of the three primary functions the system was meant to perform)! The situation presented is referred to as **task starvation**, since only the highest priority task in the system is getting CPU time and the other tasks are being *starved* of processor time.

Another detail worth pointing out is that the scheduler is still running at predetermined intervals. No matter what is going on in the system, the scheduler will diligently run at its predetermined tick rate.



There is an exception to this. FreeRTOS has a tick-less scheduler mode designed for use in extremely low power devices, which prevents the scheduler from running on the same predetermined intervals.

A more realistic use case where a preemptive scheduler is used is shown here:

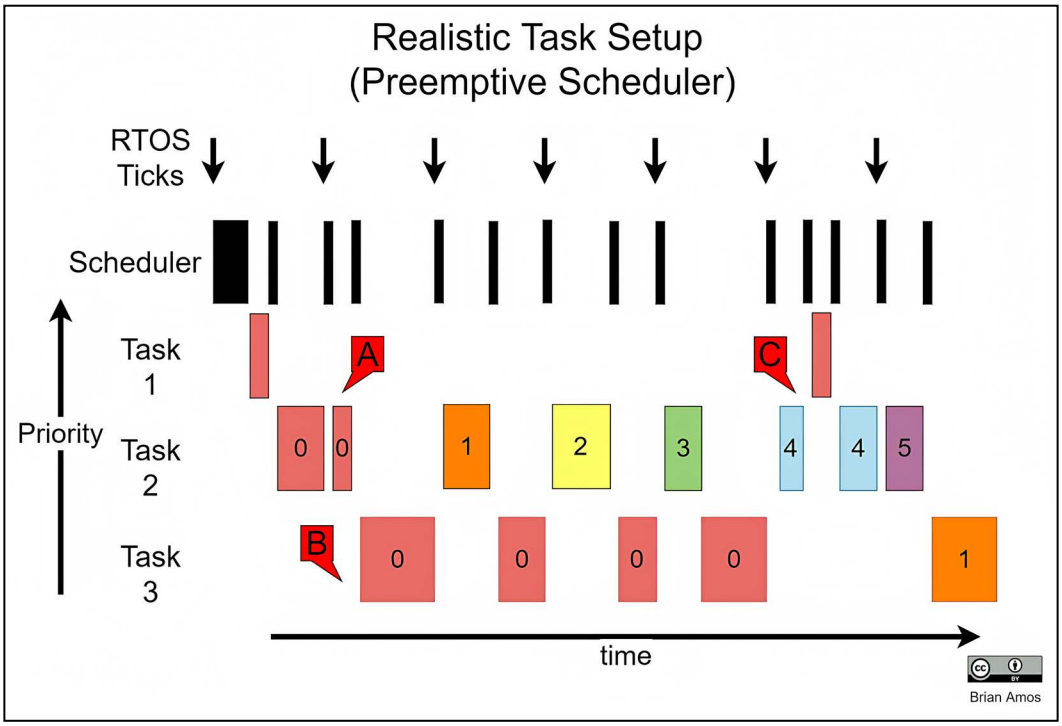


Figure 6.5 : Realistic Task Setup (Preemptive Scheduler)

In this case, `Task 1` is the highest priority task in the system (it also happens to finish executing very quickly) – the only time `Task 1` has context taken from it is when the scheduler needs to run; otherwise, it will keep context until it doesn't have any additional work to perform.

`Task 2` is the next highest priority – you'll also notice that this task is set up to execute once per RTOS scheduler tick (indicated by the downward arrows). `Task 3` is the lowest priority task in the system: it only gets context when there is nothing else worth doing in the system. There are three main points worth looking at in this diagram:

- **A**: `Task 2` has context. Even though it is interrupted by the scheduler, it immediately gets context again after the scheduler has run (because it still has work to perform and there are no other higher priority tasks needing to execute).
- **B**: `Task 2` has finished its work for iteration 0. The scheduler has run and determined that `Task 3` can have processor time. This is because no higher-priority tasks are required to run.

- **C**: `Task 2` has started running iteration 4, but `Task 1` now has some work to do – even though `Task 2` hasn't finished the work for that iteration. `Task 1` is immediately switched in by the scheduler to perform its higher priority work. After `Task 1` is finished with what it needs to do, `Task 2` is switched back in to finish iteration 4. This time, the iteration runs until the next tick, and `Task 2` runs again (iteration 5). After `Task-2` iteration-5 has completed, there is no higher priority work to perform, so the lowest priority task in the system (`Task 3`) runs again. It looks as if `Task 3` has finally completed iteration 0, so it moves on to iteration 1 and chugs along . . .

Hopefully you're still with me! If not, that's OK, given that this is a very abstract example. The key takeaway is that the highest priority task in the system takes precedence. Earlier we described how the scheduler is called by the SysTick ISR, but not how the scheduler gets called between ticks. That will be described in the next chapter: *Chapter 7*, *Running the FreeRTOS Scheduler*.



This is only a brief introduction to the relevant scheduling concepts covered in detail in the next chapter. Many more details and strategies for achieving desired system performance are discussed there, along with real-world examples.

Choosing FreeRTOS as our RTOS

There are many RTOSes that implement the schedulers discussed in this chapter. Choosing an RTOS from among them is challenging. It can be difficult, in the first place, to determine whether to use an RTOS rather than staying with bare metal. Once the decision is made to use an RTOS, it can be challenging to research the options, and choose an RTOS that meets the project's needs. Considerations that drive the decision-making process include cost, technical features, availability of source code and drivers, quality, and the team's past experience.

FreeRTOS is an open-source real-time operating system that has become popular over the years. The popularity of FreeRTOS is most likely due to two of the decision-making considerations: cost and availability of source code. FreeRTOS is free to use and open-source

In the early days of FreeRTOS, real-time operating systems were typically closed source and expensive. There were open-source options available such as Linux and eCos, but, FreeRTOS steadily gained in popularity. That was probably due, in part, to its catchy name (*Free* sounds good, doesn't it?), but it was most likely due to it being free of purchase cost and being open-source.

FreeRTOS started primarily as a simple scheduler, without all of the extra code associated with a full-blown operating system. Because FreeRTOS was open-source, a programmer could get "under the hood" to see how things worked. For those of us who were used to bare-metal programming, it seemed a simple and inexpensive way to transition to a real-time operating system. Because it was free, we could slip it into our project without needing to fill out a purchase requisition.

FreeRTOS had some other advantages besides being free and open-source. There was a commercial license option, and even a safety-certified option. This gave confidence in the quality of the code. Much open-source code is the result of a programmer's hobby time. However, FreeRTOS appeared to have the dedication of a professional team which would have been necessary to support a commercial license and safety certification. So, FreeRTOS provided the best of both worlds: high quality professional code, yet free and open-source.

Hands-on with FreeRTOS's scheduler and tasks

Below, the first section presents the chapter's FreeRTOS example-program, and the tools used to work with the program. The next section discusses the different ways FreeRTOS can be installed and configured.

Understanding and working with the example-program

The chapter's example-program is presented here. This is an introduction to using FreeRTOS and to programming with its APIs. The example-program works very similarly to the prior conceptual example, shown in *Figure 6.5*.

This section describes how the example-program works and how to run it. SystemView provides tools for monitoring and analyzing FreeRTOS execution. Instructions for using those tools are presented, and, the tools are used to explain the example-program's execution. Those tools will be used throughout the upcoming chapters.

This is an introduction to FreeRTOS programming, so not all aspects of the example-program are presented. The next chapter provides more information about the FreeRTOS scheduler and tasks. To further explore the example-program, the IDE's code-navigation features can be used, and the FreeRTOS documentation is listed in *Appendix B*.

The example-program's IDE-project

The chapter's example-program is in the book's IDE-workspace, in the project Chapter_06. The workspace and project are shown in *figure 6.6*. The example code is in `Src/main.c`. Most of the system-libraries are in folders in the workspace-directory, and not in the project-directory. In the Project Explorer, such folders are identified by the arrow in their folder icon. The folders with these system-libraries are: `BSP` (*board support packages*, i.e., for the dev-board), `Drivers` (e.g., CMSIS and HAL), and `Middleware` (e.g., FreeRTOS and SystemView). The folder `Common` is also in the workspace directory, and it has application functions that are used by multiple projects.

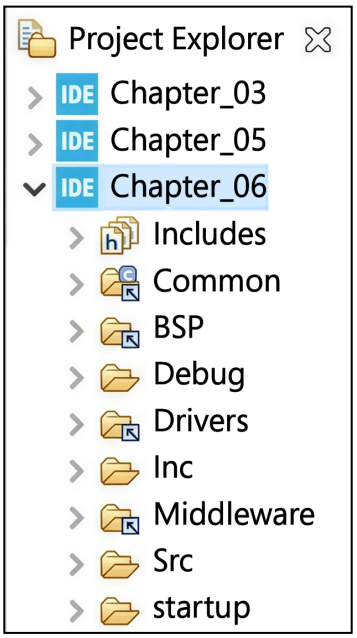


Figure 6.6 : The book's IDE-workspace

How the example-program works

The next four sections describe the chapter's example-program, and how it works. The primary parts of the program are listed here and explained. It's highly recommended that the IDE be used to further explore the program, beyond what is presented here, such as the code in `main.c`, `main()`, the functions used from `/Common`, and the FreeRTOS APIs.

Creating tasks and starting the scheduler

Open `main.c`, and we'll take a look at how it works. As shown below, three tasks are created by calling the FreeRTOS API `xTaskCreate`. The tasks themselves are functions, defined in `main.c`, i.e., `Task1`, `Task2`, and `Task3`. The tasks are assigned a priority when they are created. `Task1` has the highest priority; `Task2` has the second highest priority, and `Task3` has the lowest priority.

A task doesn't run when it is created. The FreeRTOS scheduler runs the tasks, and since we have a single processor, only one task can run at a time. If the tasks are created successfully, then the FreeRTOS scheduler is started, by calling `vTaskStartScheduler`. The scheduler will then start the highest priority task, i.e., `Task1`.

Notice `main()` has an infinite loop: `while(1)`. However, it's not a super-loop. Its purpose is to catch errors that occur in creating the tasks, or in running the scheduler. In this system, the scheduler runs forever, and it only returns if an error is encountered, such as over-running the FreeRTOS heap.

```
iterationsPerMilliSecond = lookBusyIterationRate();

if (xTaskCreate(Task1, "task1", stackSize, NULL, tskIDLE_PRIORITY + 3,
    NULL) == pdPASS)
{
  if (xTaskCreate(Task2, "task2", stackSize, NULL, tskIDLE_PRIORITY + 2,
      NULL) == pdPASS)
  {
    if (xTaskCreate(Task3, "task3", stackSize, NULL, tskIDLE_PRIORITY + 1,
        NULL) == pdPASS)
    {
      // Start the scheduler - shouldn't return unless there's a problem
      vTaskStartScheduler();
    }
  }
}

// If you've wound up here, there is likely an issue with over-running the
// FreeRTOS heap
while(1)
{
}
```

How Task1 works

Next, we'll look at how the tasks work. Again, their operation is very similar to the tasks described in the last conceptual example (*figure 6.5*). `Task1` is shown below. In the task's infinite-loop, each iteration performs 0.25 ms of "useful" processing, then the task waits for 5 SysTicks.

The function `lookBusy` is used to simulate useful processing. The function is defined in the workspace's folder `Common`. The function `lookBusy` is a simple for-loop that repeatedly performs a single arithmetic operation, to consume CPU cycles. The number of for-loop iterations is specified by `lookBusy`'s argument `lookBusyIterations`, which will be explained shortly.

The task is made to wait for 5 SysTicks by calling `vTaskDelay`, a FreeRTOS API. (Recall, a SysTick is typically 1 ms, as with this system.) When `vTaskDelay` is called, control is passed to the FreeRTOS scheduler. `Task1` will have stopped running, and it won't be eligible to run again for 5 SysTicks. Since `Task1` is being delayed, the scheduler must find another task to run. It will be the highest priority task that is eligible to run, and the scheduler will perform a context switch to that task. In this case, it will be `Task2`.

```
void Task1(void *argument)
{
  uint32_t lookBusyIterations;
  uint32_t iterationCount = 0;

  BlueLed.On();
  while(SEGGER_SYSVIEW_IsStarted()==0)
    {lookBusy(iterationsPerMilliSecond);}
  SEGGER_SYSVIEW_PrintfHost("Task1: starting\n");

  // Calculate the number of iterations for lookBusy() to spin for 1/4 of
a ms
  lookBusyIterations = iterationsPerMilliSecond / 4;

  while(1)
  {
    // SEGGER_SYSVIEW_PrintfHost is only called every 100 iterations
    iterationCount++;
    if ((iterationCount % 100) == 1)
    {
       SEGGER_SYSVIEW_PrintfHost("Task1. Iteration: %u\n", iterationCount);
```

```
    }

    // Simulate useful processing. Spin for 1/4 of a ms (processor time).
    lookBusy(lookBusyIterations);
    // Wait 5 SysTicks before resuming processing
    vTaskDelay(5);
  }
}
```

`lookBusy` is intended to spin for a particular amount of processor time. For `Task1`, we want to use 0.25 ms of processor time, so we need to calculate the respective number of for-loop iterations. Toward this end, the function `lookBusyIterationRate` is provided to calculate the number of for-loop iterations in 1 ms. That function is called in `main()`, just before the tasks are created, as shown above. The function's return value is stored in the global variable `iterationsPerMilliSecond`. When `Task1` is first entered, `iterationsPerMilliSecond` is used to calculate the number of for-loop iterations in 0.25 ms.

Starting SystemView

When `Task1` is entered, another thing it does is to wait for SystemView's Record-mode to be started. SystemView has excellent features for monitoring and debugging FreeRTOS operations, which we will be using. When SystemView is used with FreeRTOS, SystemView requires that Record-mode be started after the FreeRTOS scheduler, and not before then. An additional requirement is that Record-mode must be started before the firmware starts writing SystemView messages. This is just a practical requirement, to ensure messages are not lost. The messages are written by calling `SEGGER_SYSVIEW_PrintfHost` (similar to `printf`).

`Task1` provides a user-interface for starting Record-mode after the scheduler starts, and before SystemView messages are generated. The blue LED is turned-on when `Task1` is entered, and it indicates Record-mode can be started. `Task1` will then spin until Record-mode is started:

```
while(SEGGER_SYSVIEW_IsStarted()==0){lookBusy(iterationsPerMilliSecond);}
```

`SEGGER_SYSVIEW_IsStarted` communicates with the SystemView app on the development-computer, to ask if Record-mode is running. For each loop iteration, lookBusy is run for 1 ms. That throttles the frequency of calls to `SEGGER_SYSVIEW_IsStarted`.

We also need to ensure the other tasks do not call `SEGGER_SYSVIEW_PrintfHost` before Record-mode is started. Since `Task1` is the highest priority task, the other tasks will not run until after `Task1` runs `vTaskDelay`, which is well after Record-mode was started.

The two SystemView APIs that we use involve communication between SystemView on the dev-board and SystemView on the development computer. The communication bandwidth is limited, as well as buffers used to hold incoming and outgoing data. If too much data is sent, the excess will be dropped, and SystemView will report an overflow. Thus, the calls to the SystemView APIs have to be throttled. In `Task1`, the infinite loop has a call to `SEGGER_SYSVIEW_PrintfHost`. However, it's only called every 100 iterations, to avoid overflow. For the throttled API calls, the amount of throttling needed was determined by testing.

How Task2 and Task3 work

`Task2` and `Task3` work similarly to `Task1`. The main differences are the amount of processor time used by `lookBusy`, and how long the delay is with `vTaskDelay`. `Task2` is shown below. `lookBusy` runs for 0.5 ms of processor time. `vTaskDelay` waits for 1 SysTick. `Task3` is identical, but there, `lookBusy` runs for 2 ms of processor time and, `vTaskDelay` is not called—there is no wait.

There's a difference in the timing effects from `lookBusy` and `vTaskDelay`. `lookBusy` runs for a particular amount of processor time. When `lookBusy` is running, a context switch to another task will increase `lookBusy`'s duration in wall-clock time, but the context switch will not affect how much processor time `lookBusy` ultimately uses. On the other hand, the duration specified for `vTaskDelay` is in SysTicks (with 1 ms between ticks). The duration starts when vTaskDelay is executed, and it ends in X SysTicks. At the end of that duration, the task becomes eligible to continue running, as determined by the scheduler.

```
void Task2( void* argument )
{
  uint32_t lookBusyIterations;
  uint32_t iterationCount = 0;

  GreenLed.On();
  SEGGER_SYSVIEW_PrintfHost("Task2: starting\n");

  // For Task2, lookBusy() needs to spin for 1/2 of a ms (processor time).
  lookBusyIterations = iterationsPerMilliSecond / 2;

  while(1)
  {
    iterationCount++;
    if ((iterationCount % 100) == 1)
    {
      SEGGER_SYSVIEW_PrintfHost("Task2. Iteration: %u\n", iterationCount);
```

```
    }

    // Simulate useful processing. Spin for 1/2 of a ms  (processor time).
    lookBusy(lookBusyIterations);
    // Wait 1 SysTick before resuming processing
    vTaskDelay(1);
  }
}
```

Running the example-program

This section, and the next section, describe how to run the example-program, including how to use SystemView to view the FreeRTOS operation. The example-program is run using the tools presented in chapter 3: STM32CubeIDE, Ozone and SystemView. The first two tools are used in much the same way as before, and differences are described here. SystemView use is described in the next section.

First, use STM32CubeIDE to build the program. In the Project Explorer, right click on the **Chapter_06** project, and select **Build**. The project should compile with no errors.

Ozone will be used to load the firmware on the dev-board. A `.jdebug` file will be needed. The file `Ozone-project-file.jdebug` can be copied from any of the earlier chapters' IDE-project, or from the Templates folder in the workspace. Configure the file as described in its top comment-block. Ensure the line for `Project.SetOSPlugin()` is un-commented. That API is needed when FreeRTOS is used.

Next, use Ozone to load the firmware on the dev-board and start it. The blue LED should turn on, which indicates Task1 is running and waiting for SystemView Record-mode to be started.

Using SystemView with FreeRTOS

SystemView records the FreeRTOS execution in real time, including when the scheduler runs, and when tasks are stopped and started. The SystemView software on the MCU sends this execution data to the SystemView app on the development computer. With several tasks running, it's easy to send more data than our debug-probe can handle. When that happens, the excess data is dropped and SystemView will report an overflow condition, as shown in *figure 6.7*. (Additional information on SystemView overflow is in *Chapter 16*, *Troubleshooting Tips and Next Steps* .)

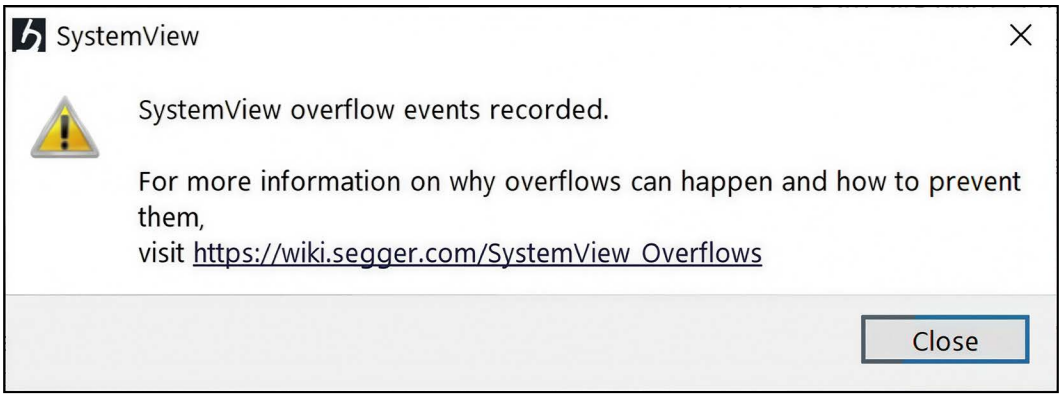


Figure 6.7 : SystemView's overflow message

Fortunately, there is a simple solution to this overflow problem: **before starting SystemView Record-mode, first stop the Ozone debug-session, if FreeRTOS is being used**. In our experiments, this solution has usually been sufficient for the book's FreeRTOS programs. The primary source of the problem may be the the dev-board's J-Link debug-probe, and its relatively slow SWD target-interface speed of 4 MHz. In contrast, Segger's other debug-probes have speeds of up to 50 MHz.

So, for firmware that runs FreeRTOS, after starting the firmware with Ozone, stop the Ozone debug-session. This can be done by clicking on Ozone's red on/off icon, or by clicking on **Debug : Stop Debug Session**. SystemView Record-mode can then be started.



Before starting SystemView Record-mode, first stop the Ozone debug-session, if FreeRTOS is being used.

Going back to running our example-program, after the blue LED turns on, stop the Ozone debug-session. Then, start the SystemView app, and start its Record-mode. Wait a few seconds, and stop the recording. There's now enough data collected to see the FreeRTOS operation.

We'll use SystemView to explore FreeRTOS's scheduling and task execution. We'll see that there is an execution pattern which repeats every 5 SysTicks (5 ms). And, we'll examine one period of the pattern, in depth.

First, ensure timestamps are displayed in the format called "Recording Time", as shown in *figure 6.8*. This format sets all timestamps to be relative to the first recorded event, which occurs at time 0.

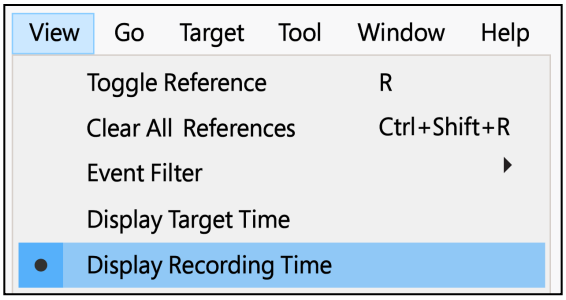


Figure 6.8 : Setting SystemView's timestamp format

Next, we'll look at the SystemView window, and two panes that are normally at the top. The right-most pane is called the **Terminal** window, and it's shown in *figure 6.9*. (If it isn't open, click on **Window : Terminal**). The Terminal window shows the messages created by `SEGGER_SYSVIEW_PrintfHost`. (If the window shows additional columns, they can be closed by right-clicking on the column header.)

In the Terminal window, click on one of the messages generated by `Task1`, but not its first message, as illustrated in *figure 6.9*. This message will be at the beginning of the repeating pattern described earlier.

(The following figures are from running the tasks with messages generated every 10 loop-iterations. In the code presented earlier, the tasks generated messages every 100 loop-iterations.)

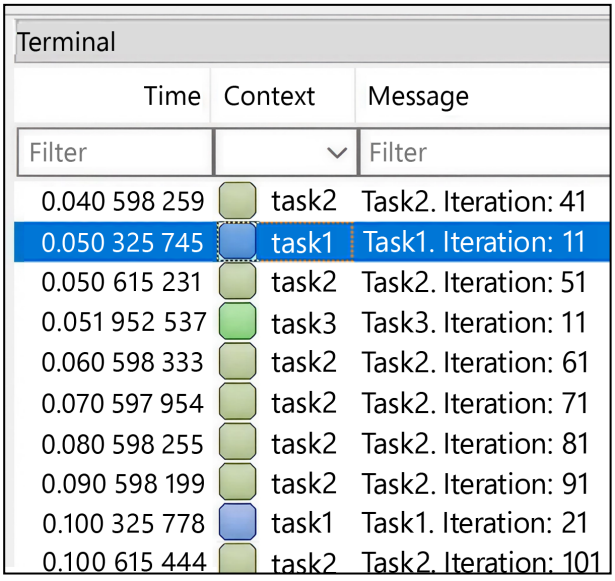


Figure 6.9 : The SystemView Terminal window

The left-most pane is called the **Events List** window, and it is shown in *figure 6.10*. It shows all of the events recorded by SystemView, including the messages from `SEGGER_SYSVIEW_PrintfHost`. After clicking on the `Task1` message in the Terminal window, its message-event should also be visible in the Events List window. The message-event can be located using the message's timestamp in the Terminal window. In the Events List window in *figure 6.10*, it is event #369. Position the window to also display the SysTick events that precede the Task1 message. Here, they are events #365-367, as shown in *figure 6.10*.

#	Time	Context	Event	Detail
364	0.049 826 352	task3	Task Run	
365	0.050 288 296	SysTick	ISR Enter	
366	0.050 294 162	SysTick	Task Ready	task1, runs after 14.653 us
367	0.050 299 912	SysTick	Task Ready	task2, runs after 298.417 us
368	0.050 308 815	task1	Task Run	Runs for 278.954 us
369	0.050 325 745	task1	Log	Task1. Iteration: 11
370	0.050 582 338	task1	vTaskDelay	xTicksToDelay=5
371	0.050 587 769	task1	Task Block	Delayed
372	0.050 598 329	task2	Task Run	Runs for 529.171 us
373	0.050 615 231	task2	Log	Task2. Iteration: 51
374	0.051 121 991	task2	vTaskDelay	xTicksToDelay=1
375	0.051 127 500	task2	Task Block	Delayed
376	0.051 138 065	task3	Task Run	
377	0.051 288 269	SysTick	ISR Enter	
378	0.051 294 144	SysTick	Task Ready	task2, runs after 9.583 us
379	0.051 303 727	task2	Task Run	Runs for 511.838 us
380	0.051 810 051	task2	vTaskDelay	xTicksToDelay=1
381	0.051 815 565	task2	Task Block	Delayed
382	0.051 826 102	task3	Task Run	

Figure 6.10 : The SystemView Events-List window

The **Events List** widow has a column named **Context**, and its cells contain a colored box followed by the context's name, e.g., task1, SysTick, etc. The boxes' color-patterns include a solid color, or two colors. A solid-colored box indicates that only the named-context is running at the timestamp, e.g., event # 366. Two colors indicate there is a transition from the context named in the prior row, to the context named in the current row, e.g., event #365.

The **Timeline** window displays FreeRTOS's operation, and it is shown in *figure 6.11*. (If the window isn't open, click on **Window : Timeline**.) The Timeline window is usually at the bottom of the SystemView window. As illustrated in *figure 6.11*, the rows show the system activity over time, and the activity is organized by context, i.e., SysTick timer, scheduler, task, interrupt, and idle. The rows are organized by scheduling priority, with the highest priority at the top. We'll refer to the Timeline window as simply the *Timeline*.

From the Timeline we can see there is a pattern to the tasks' execution: `Task1` executes, followed by `Task2` and `Task3` each executing five times. The pattern repeats every 5 ms.

Notice that some events are missing in the window, e.g., the SysTick event occurs every ms, but it's only shown at 5 ms. This is because the window was not wide enough to show all of those events. The window can be made bigger by dragging its edges. When using the Timeline, keep in mind that some events might not be shown.

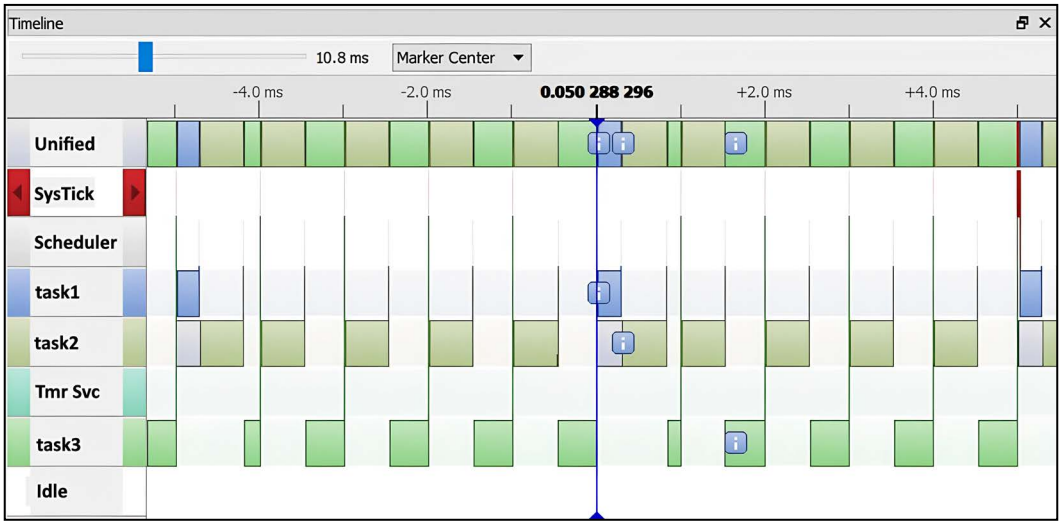


Figure 6.11 : The SystemView Timeline window

Figure 6.12 shows the Timeline's controls and components. The **Time axis** specifies the timescale used, and here it's in micro-seconds. The **Zoom level** is selected via the slide bar. The zoom level specifies the number of seconds in the time axis's range. In *figure 6.11*, the time axis's range is set to 10.8 ms. So, 10.8 ms of events are shown. The time-axis's vertical marks range from -5.0 ms to 5 ms, a span of 10 ms. In *figure 6.12*, the time axis's range is set to 3.42 ms. However, the figure is cropped, so it does not show the whole range.

In *figure 6.12*, the **Marker** is the blue vertical line. By clicking on an event in the Event List, the event is displayed in the Timeline, at the marker. In the figure, the marker was set here by clicking on event #365, in the Event List shown in *figure 6.10*. The **Marker timestamp** is the marked-event's timestamp.

The **Marker position** specifies the marker's relative position in the time axis's range. In *figure 6.11*, the marker is set to the middle of the time axis's range. It is set by using the Marker position drop-down box, and selecting **Marker Center**. In *figure 6.12*, the marker is set at 10% of the time axis's range (again, the figure has been cropped). It is set by selecting **Marker at 10%**, from the drop-down box.

In the time axis, the units are specified relative to the marker position. As in *figure 6.11*, units to the left of the marker are negative, and they are positive to the right.

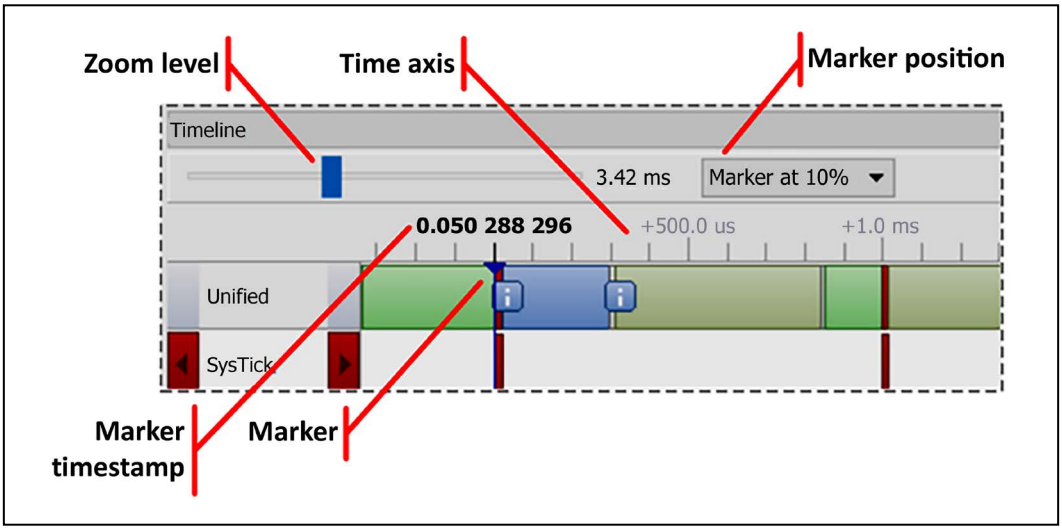


Figure 6.12 : The Timeline's controls and components

Next, we'll display specific parts of the Event List in the Timeline. We'll continue the example shown earlier for the Terminal window and Event List. In this case, we'll show one instance of the pattern, starting with `Task1`.

In *figure 6.13*, the marker is at event #365, and at the 10% position in the time range. (The figure is expanded in *figures 6.13a* and *6.13b*.) The time range is set to 5.84 ms, to display the 5 ms pattern. For SystemView to display all of the context data, the Timeline window must be made sufficiently wide. The width can be adjusted by using the mouse to drag the window's left edge.

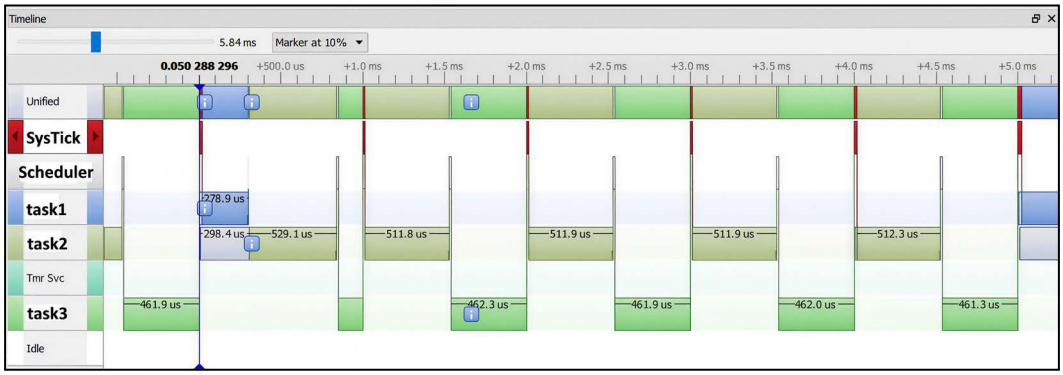


Figure 6.13 : The 5 ms pattern for task execution

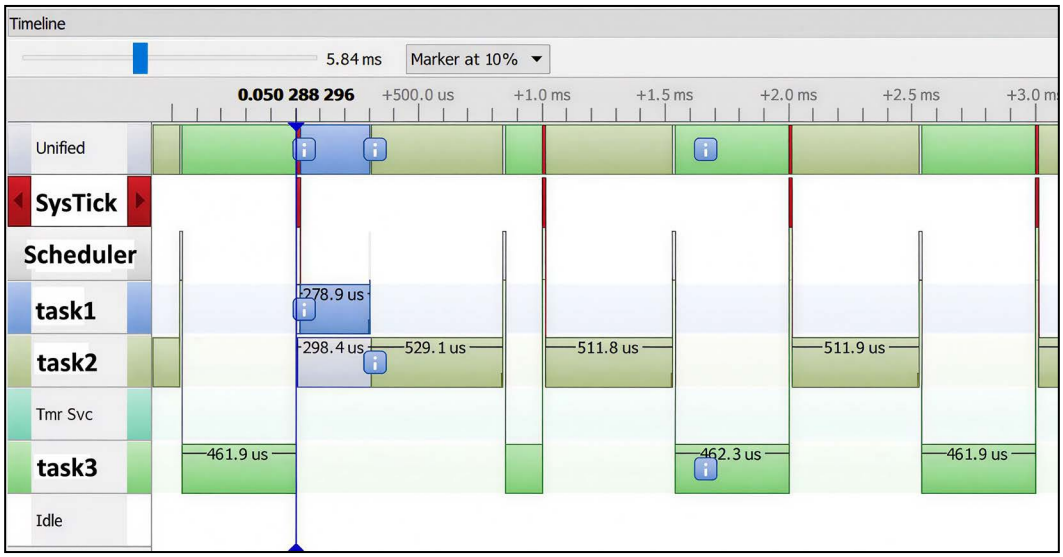


Figure 6.13a : The left side of figure 6.13

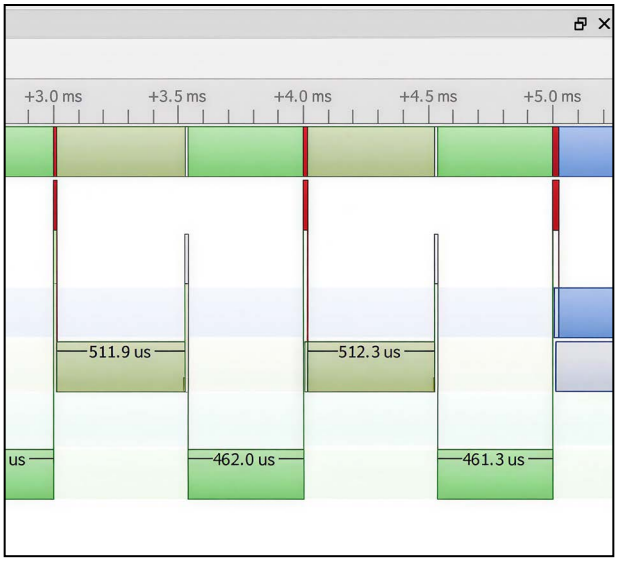


Figure 6.13b : The right side of figure 6.13

For events #365-367, the SysTick ISR runs, as well as the scheduler. The timeline shows SysTick runs every 1 ms, and `Task1` runs again after 5 SysTick events. This verifies `Task1`'s use of `vTaskDelay` for 5 SysTicks.

Event #364 shows `Task3` was running before the SysTick ISR, and that is also shown in the timeline. `Task1` runs for 278.9 us, as shown in the Detail field for Event #368, and in the timeline. `Task1` calls `lookBusy` to run for 250 us, and the code appears to be working properly, given that `Task1` ran for 278.9 us.

In `Task1`, after `lookBusy` runs, `vTaskDelay` is then called. `vTaskDelay` causes the scheduler to run, as seen in Event #370-371, and in the timeline. `Task2` is the highest priority task that is available run. The scheduler performs a context switch to run `Task2`, as seen in Event #372, and in the timeline.

`Task2` runs for 529.1 us, as seen in Event #372, and the Timeline. This is consistent with `Task2`'s calculations to run `lookBusy` for 500 us. When `lookBusy` completes, `Task2` runs `vTaskDelay` for a one SysTick delay, as can be seen in events #374-375. `vTaskDelay` causes the scheduler to run, and the scheduler performs a context switch to run `Task3`, which is the only task available. `Task3` runs, as shown in Event #376, and the Timeline.

Next, the SysTick ISR starts, as shown in Event #377, and in the timeline. `Task2`'s delay ends as shown in Event #378. The scheduler determines that `Task2` is now the highest priority available task. The scheduler then performs a context switch to run `Task2`, as shown in Event #379.

`Task2` and `Task3` continue in this manner, with alternating use of the processor, until `Task1`'s delay ends. That can be seen in the timeline, at +5.0 ms.

In the Timeline, `Task2` is colored grey, just after the marker. For time-intervals that are colored grey, it indicates a task was made ready to run, but a higher priority task is running instead. At the marker, `Task2`'s delay ended, making it ready to run (event #367). However, `Task1` was also ready to run, and it had higher priority.

This concludes the chapter's introduction to using FreeRTOS and to programming with its APIs.

Installing and configuring FreeRTOS

The first section discusses installing FreeRTOS, and a second section discusses configuring FreeRTOS.

Installing FreeRTOS

The best way to learn the material presented in this book is to augment study with hands-on experiments and development. To do that, an IDE-project is needed with FreeRTOS installed. This section discusses how FreeRTOS can be installed in an IDE-project.

There are a few options for installing FreeRTOS in an IDE-project. The IDE can be configured to generate FreeRTOS's source-code in the project. Another option is to manually install FreeRTOS. A third option is to import an IDE-project that is configured for the target hardware and that has FreeRTOS installed. The book's code repository has an IDE-project like that, which can be imported for use on our dev-board. It is named `FreeRTOS_project`. It's available in the example-programs' workspace.

The easiest way to install FreeRTOS is to configure the IDE to generate FreeRTOS source-code. However, this approach has a number of limitations, making it best-suited for simple prototypes. For example, any modifications to the FreeRTOS source-code will get overwritten when the system configuration-file (*.ioc) is changed. For instance, using SystemView with FreeRTOS requires patching the FreeRTOS source code, and those changes would be overwritten. Another limitation is that the generated FreeRTOS source-code is usually not the latest version. A further limitation is that the generated `main.c` has API calls that create at least one FreeRTOS task and that start the FreeRTOS scheduler. However, instead of using FreeRTOS APIs, CMSIS-OS APIs are used instead. CMSIS-OS is a wrapper for FreeRTOS.

Typically, the preferable way to use FreeRTOS is to manually install it in the IDE-project. This avoids the problems with using the generated FreeRTOS source-code, that were just described. The disadvantage of this approach is that it can require non-trivial research. The installation process varies depending on the MCU used, and the FreeRTOS configuration-requirements.

All three of these FreeRTOS installation-techniques are presented in detail, in *Appendix C, Creating FreeRTOS-projects, and installing FreeRTOS*.

Configuring FreeRTOS

Most of the book's example-programs use FreeRTOS, starting with the present chapter. As described earlier, for those IDE-projects, most of the system-libraries reside in folders in the workspace-directory. This avoids having redundant copies of the system-libraries in each IDE-project. Accordingly, most of the FreeRTOS source-code is in the workspace-directory, in the folder `Middleware\Third_Party\FreeRTOS`. This folder is used by the chapters' example-programs that use FreeRTOS.

(The prior section described the IDE-project `FreeRTOS_project`. It contains its own FreeRTOS source-code, and it does not use the FreeRTOS source-code in `Middleware\Third_Party\FreeRTOS`.)

FreeRTOS has a config-file `FreeRTOSConfig.h`. For the IDE-projects that use FreeRTOS, that config-file is in the project-directory at `Inc/FreeRTOSConfig.h`. Each IDE-project has it's own copy of the file. Though, there is little difference among those files.

For the FreeRTOS config-file, its settings are numerous, and some settings require an understanding of complex FreeRTOS internals. The settings are documented in the FreeRTOS reference manual, and on the FreeRTOS web-site. The documentation on the web-site is more complete. (*Appendix B* lists the relevant FreeRTOS documentation and where to get it.) Some config-file settings are not documented, and the source-code must be consulted for them.

Some of the configuration settings are related to the scheduling concepts presented earlier in the present chapter. Those settings are briefly described here, to show the concepts' practical application.

We described a preemptive scheduling policy where higher-priority tasks are run before lower-priority tasks. In FreeRTOS, the use of preemptive scheduling is specified by the configuration-setting `configUSE_PREEMPTION`. Setting it to 1 enables the feature, and 0 disables it. The default setting is 1. In the book's IDE-projects, the `FreeRTOSConfig.h` files enable preemptive scheduling:

```
#define configUSE_PREEMPTION  1
```

We also described a round-robin scheduling policy where each task is given a time-slice to run between each scheduler tick. The configuration-setting `configUSE_TIME_SLICING` pertains to time-slicing of equal priority tasks (round-robin scheduling). Setting it to 1 enables the feature, and 0 disables it. The default setting is 1. In the book's IDE-projects, the `FreeRTOSConfig.h` files do not define `configUSE_TIME_SLICING` , so the feature is enabled by default.

There is a combination of these two configuration-settings in which tasks of higher priority are caused to be run by the scheduler over lower-priority tasks (pre-emption) and tasks of equal priority are given a time-slice in a round-robin fashion. This is the default configuration for the FreeRTOS scheduler as it is used when both `configUSE_PREEMPTION` and `configUSE_TIME_SLICING` are omitted.

Appendix C describes how to install FreeRTOS. It has some additional information on configuring FreeRTOS, and on `FreeRTOSconfig.h`.

Summary

In this chapter, we explored the need for something more than a super-loop, when a system has complex time constraints. RTOS tasks were presented as a solution for such systems, and they were compared to super-loop functions. The task-based architecture was introduced from a conceptual view, including the two primary types of scheduling used in FreeRTOS: round-robin and preemptive. Examples were provided, from a conceptual perspective, to show how a preemptive-scheduler runs tasks with different priorities.

A FreeRTOS program was then presented, which implements the preemptive-scheduling concepts. Three tasks were created, with different priorities, and they were run by FreeRTOS's preemptive scheduler. Also presented were SystemView's tools for monitoring FreeRTOS-based systems.

The best way to learn the material in this book is to augment one's study with hands-on experiments and development. In order to do that, an IDE-project is needed, with FreeRTOS installed. Three options for creating a FreeRTOS IDE-project were described, and *Appendix C* provides detailed instructions. A FreeRTOS IDE-project is included along with the example-programs, and it's named `FreeRTOS_project`.

In the next chapter, we'll look at tasks in more depth, including their creation, configuration, operating states, and troubleshooting. Real-world examples will be presented, via example-programs. The programs show code that's needed for creating reliable real-time systems.

7

Running the FreeRTOS Scheduler

The prior chapter introduced the FreeRTOS scheduler and tasks. We also introduced the use of our software tools for developing FreeRTOS-based systems, including STM32CubeIDE, Ozone, and SystemView. This chapter continues those topics, in more depth.

The FreeRTOS scheduler takes care of all task-switching decisions. The most basic things you can do with an RTOS include creating a few tasks and then starting the scheduler – which is exactly what we'll be doing in this chapter. Creating tasks and getting the scheduler up and running will become something you'll be well accustomed to after some practice. Even though this is straightforward, it doesn't always go smoothly (especially on your first couple of tries), so we'll also cover some common problems and how to fix them. By the end, you'll be able to set up your own RTOS application from scratch and know how to troubleshoot common problems.

We'll start by covering two different ways of creating FreeRTOS tasks and the advantages each offers. From there, we'll cover how to start the scheduler and what to look for to make sure it is running. These topics are illustrated by an example-program. We'll run the program using our development tools, and show more of their FreeRTOS features.

Building on our knowledge gained from the example-program, we'll further explain the scheduler and how it works. Topics on task implementation and troubleshooting are also presented.

This chapter presents two example-programs. They are in the book's IDE workspace, in the project `Chapter_07`. The first example-program presents the topics related to the FreeRTOS tasks and scheduler. The second example-program presents start-up problems, and how to troubleshoot them.

The system documentation for the chapter is listed in *Appendix B*. The system documentation is voluminous, and downloading it piecemeal, with each chapter, makes that chore manageable.

The chapter's sections are:

- Creating tasks and starting the scheduler
- Deleting tasks
- Trying out the code
- Understanding the scheduler
- Task implementation and troubleshooting

Creating tasks and starting the scheduler

The chapter's first example-program is a FreeRTOS application with three tasks. Each task blinks an LED, and it interacts with the other tasks. The interactions are designed to illustrate particular ways that FreeRTOS works.

The FreeRTOS-based application is in the IDE-project `Chapter_07`, in the file `Src\main_taskCreation.c`. We'll show how the program works, then run the program, and use SystemView to analyze the tasks' behavior.

In order to get an RTOS application up and running, a few things need to happen:

1. The MCU hardware needs to be initialized.
2. The task functions need to be defined, e.g., C functions.
3. RTOS tasks need to be created and mapped to the task functions.
4. The return values for task-creation need to be checked, and errors handled.
5. The RTOS scheduler must be started.

These are the typical steps for using the scheduler and tasks. Of course, there are additional task-related functions that can be performed, such as deleting tasks, and creating tasks after the scheduler is started.

Our example-program uses these five steps, and we'll break down each of them.

Initializing hardware

Before we can do anything with the RTOS, we need to make sure that our hardware is configured properly. This will typically include carrying out activities such as ensuring GPIO lines are in their proper states, configuring external RAM, configuring critical peripherals and external circuitry, performing built-in tests, and so on. For the example-programs in *Chapter 6*, and later, the MCU **hardware initialization** can be performed by calling `HWInit()`. It performs all of the basic hardware initialization required:

```
int main(void)
{
    HWInit();
```

Next, let's define the task behavior we'll be programming and take a look at what our individual task functions look like.

Defining task functions

The three tasks are named `GreenTask`, `BlueTask`, and `RedTask`, after the LED that each of them blinks. Each task is implemented as a **C function**, and the function's name is the task's name. The functions are defined in `main_taskCreation.c`. The functions' code is shown below and explained. Readers are encouraged to follow along in the IDE, and further explore the code there.

Recall that a task is really just an infinite `while` loop with its own stack and a priority assigned to it. Let's cover the tasks one by one.

`GreenTask` is shown below and explained. `GreenTask` sleeps for a little while (1.5 seconds) with the green LED on, and then deletes itself.

```
void GreenTask(void *argument)
{
    BlueLed.On();
    while(SEGGER_SYSVIEW_IsStarted()==0){
        lookBusy(iterationsPerMilliSecond);
    }
    BlueLed.Off();

    SEGGER_SYSVIEW_PrintfHost("GreenTask started");

    GreenLed.On();
```

```
    vTaskDelay(1500 / portTICK_PERIOD_MS);
    GreenLed.Off();

    SEGGER_SYSVIEW_PrintfHost("GreenTask is deleting itself");
    vTaskDelete(NULL);

    // The task never gets to here
    GreenLed.On();
}
```

There are a few noteworthy items in `GreenTask`:

- The `while` loop spins until the user starts the SystemView app in Record mode. This synchronization scheme was described in the prior chapter. The blue LED indicates the loop is spinning.
- The *sleep* is implemented by the FreeRTOS API `vTaskDelay`. It causes the task to stop running. The task is eligible to be restarted by the scheduler in 1.5 seconds, as will be explained later.
- `vTaskDelay`'s input is the number of ticks to delay. As described in the prior chapter, ticks are generated every milli-second by the SysTick timer. It is a countdown timer, and when it reaches zero, its **ISR** is called. The ISR calls the scheduler. The moment the SysTick timer reaches zero is referred to as the *tick*.
- vTaskDelay's argument is `(1500 / portTICK_PERIOD_MS)`. It converts 1,500 ms to the comparable number of ticks. On our system, the tick period is 1 ms, so `portTICK_PERIOD_MS` is 1 ms/tick. And, 1,500 ms converts to 1,500 ticks. (To be precise, the first tick will occur in less than one tick period, since `vTaskDelay` is called at some point between the first tick and the prior tick. So, the actual delay is somewhere between 1,499 and 1,500 ms.)
- Normally, a task will contain an infinite `while` loop so that it doesn't return. Such a `while` loop isn't needed here, since the task deletes itself.
- You can confirm `vTaskDelete` doesn't allow execution past the function call by looking at the dev-board. The green LED will only be on for 1.5 seconds before shutting off permanently.

`BlueTask` is shown below. It blinks the blue LED rapidly for an indefinite period of time, thanks to the infinite `while` loop. However, the blue LED blinks are cut short because `RedTask` will eventually delete `BlueTask`.

```
void BlueTask( void* argument )
{
    while(1)
    {
        SEGGER_SYSVIEW_PrintfHost("BlueTask is starting a loop
iteration");
        BlueLed.On();
        vTaskDelay(200 / portTICK_PERIOD_MS);
        BlueLed.Off();
        vTaskDelay(200 / portTICK_PERIOD_MS);
    }
}
```

`RedTask` is shown below. Two types of delay are used in blinking the red LED. The `for`-loop runs for 1 second of processor time. (`lookBusy` was described in the prior chapter.) The calls to `vTaskDelay` each take at least 0.5 seconds of wall-clock time. On the first `while` loop iteration, `BlueTask` is deleted. After that, the red LED will blink with a pattern of approximately 1.5 seconds off, and 0.5 seconds on.

```
void RedTask( void* argument )
{
    . . .
    while(1)
    {
        SEGGER_SYSVIEW_PrintfHost("RedTask is starting a loop iteration");
        for (i=0;i<1000;i++){
            lookBusy(iterationsPerMilliSecond);
        }

        SEGGER_SYSVIEW_PrintfHost("RedTask is turning-on the red LED");
        RedLed.On();
        vTaskDelay(500/ portTICK_PERIOD_MS);
        RedLed.Off();
        vTaskDelay(500/ portTICK_PERIOD_MS);

        if(firstIteration == 1)
        {
            SEGGER_SYSVIEW_PrintfHost("RedTask is deleting BlueTask");
```

```
            vTaskDelete(blueTaskHandle);
            firstIteration = 0;
        }
    }
}
```

Note that among these tasks, they either run forever, or are explicitly deleted. A key requirement for FreeRTOS tasks is that they must not be allowed to return from their implementing function in any way. Tasks cannot contain a `return` statement, and they cannot be permitted to execute past the end of the function. If a task is no longer needed, it needs to be explicitly deleted.



FreeRTOS tasks must not be allowed to return from their implementing function in any way.

If a task is no longer needed, it needs to be explicitly deleted.

So, the preceding functions don't look like anything special – and they're not. They are simply standard C functions, two of which have infinite `while` loops in them. How do we go about creating FreeRTOS tasks out of these plain old functions?

Creating tasks

In our example-program, the tasks are created in `main()`, as shown below. Two of the tasks are created with **xTaskCreate**, which is described here. The other task is created with `xTaskCreateStatic`, which is described later; it works just like `xTaskCreate`, but uses static memory allocation instead of the heap. Typically, all tasks would be created using the same API, but they are mixed in our example, for demonstration purposes.

After the tasks are created, the scheduler is started. The scheduler runs indefinitely, and should not return unless there is a problem, such as over-running the FreeRTOS heap. In `main()`, the infinite `while` loop is used to catch such **errors** in the scheduler.

```
if ( xTaskCreate(GreenTask, "GreenTask", STACK_SIZE, NULL,
                 tskIDLE_PRIORITY + 2, NULL ) != pdPASS){ while(1); }

assert_param( xTaskCreate(BlueTask, "BlueTask", STACK_SIZE, NULL,
                          tskIDLE_PRIORITY + 1, &blueTaskHandle) == pdPASS );

xTaskCreateStatic(RedTask, "RedTask", STACK_SIZE, NULL,
                  tskIDLE_PRIORITY + 1, RedTaskStack,
```

```
                    &RedTaskTCB);

vTaskStartScheduler();
while(1)
{
}
```

In FreeRTOS, there are requirements for when tasks can be created, relative to starting the scheduler:

- FreeRTOS requires that at least one task be created before starting the scheduler.
- Because the scheduler doesn't return, it is not possible to start a task from `main()` after starting the scheduler.
- Once the scheduler is started, tasks can create new tasks as necessary.

Here's what the prototype for FreeRTOS task-creation looks like:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        configSTACK_DEPTH_TYPE usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask);
```

A call to the function would look like this:

```
retVal = xTaskCreate(Task1, "task1", StackSizeWords, NULL,
                     tskIDLE_PRIORITY + 2, tskHandlePtr);
```

This function call might be a little longer than expected – let's break it down:

- `Task1`: The name of the function that implements the task.
- `"task1"`: The task's name. It is used by tools like Ozone and SystemView, when displaying the task.
- `StackSizeWords`: The number of **words** reserved for the task's stack.
- `NULL`: A pointer that can be passed to the underlying function. The pointer must still be valid when the task runs and references the pointer.
- `tskIDLE_PRIORITY + 2`: This is the priority of the task being created. This particular call is setting the priority to two levels higher than the priority of the **Idle task** (which runs when no other tasks are running).

- `tskHandlePtr`: This is a pointer to a `TaskHandle_t` data type. This **task handle** can be passed to other tasks that reference the present task, e.g., to delete the present task.
- `retVal`: For this call, `xTaskCreate` will return either `pdPASS` or `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY`, depending on whether or not heap space was successfully allocated. **You must check this return value!**

Note the task's **stack-size** is specified in words, and not in bytes. FreeRTOS calculates the stack size as a multiple of the type `StackType_t`. In our Cortex-M processor, `StackType_t` is `uint32_t`, and thus the word size is four bytes. So, the task's stack-size in bytes will be `4 * usStackDepth`. In the example-program, the calls to `xTaskCreate` specify the stack size as `STACK_SIZE`, which is 128 words. This is a minimum recommended stack-size for our processor.

```
#define STACK_SIZE 128
```

The FreeRTOS reference manual has a section on `xTaskCreate`, and it describes the stack-size at a high level. FreeRTOS's use of `StackType_t` and our processor's word-size were discovered by examining the FreeRTOS source code, starting with `xTaskCreate` in `task.c`. More information on the stack-size is in *Chapter 14*, *FreeRTOS Memory Management*.



FreeRTOS requires that at least one task be created before starting the scheduler.

Because the scheduler doesn't return, it is not possible to start a task from `main()` after starting the scheduler.

Once the scheduler is started, tasks can create new tasks as necessary.

Now that we've got a good idea of what the input parameters are for creating a task, let's take a look at why it's so important to check the return value.

Checking the return value

In `main()`, the scheduler should only be started if the tasks were created successfully. So, the calls to `xTaskCreate` need to be verified by checking their return value. Of course, there are many ways to accomplish this, and two of them are used in the code:

1. The first way is by wrapping the call in an `if` statement with an in-line infinite `while` loop:

```
if( xTaskCreate(GreenTask, "GreenTask", STACK_SIZE, NULL,
                tskIDLE_PRIORITY + 2, NULL) != pdPASS){ while(1) }
```

2. The second is by using an **assert** rather than the infinite `while` loop. If your project has assert support, then it would be better to use an assert, rather than an infinite `while` loop. Since our project has the HAL library included, we can make use of the `assert_param` macro:

```
retVal = xTaskCreate(BlueTask, "BlueTask", STACK_SIZE, NULL,
                     tskIDLE_PRIORITY + 1, &blueTaskHandle);
assert_param(retVal == pdPASS);
```

assert_param is an STM-supplied macro that checks whether a condition is true. If the condition evaluates as false, then `assert_failed` is called. (More information is in the HAL user manual.) `assert_failed` is a user-defined function, and we put it in `\BSP\Nucleo_F767ZI_Init.c`. It is shown below, and explained:

```
void assert_failed(uint8_t *file, uint32_t line)
{
    BlueLed.On();
    RedLed.On();
    GreenLed.On();
    SEGGER_SYSVIEW_PrintfHost("assert failed. Line: %u.  File:", line);
    SEGGER_SYSVIEW_PrintfHost((char *) file);
    while(1);
}
```

assert_failed turns on all of the user LEDs, and it uses the API `SEGGER_SYSVIEW_PrintfHost` to display the file-name and line-number for the call to `assert_param`. The API `SEGGER_SYSVIEW_PrintfHost` does not support the format-placeholder `%s`, like `printf` does. (More information is in *Chapter 16, Troubleshooting Tips and Next Steps*.) Thus, the file name is displayed via a separate call to `SEGGER_SYSVIEW_PrintfHost`. The function `assert_failed` then enters an infinite `while` loop.

When FreeRTOS is being used, if `SEGGER_SYSVIEW_PrintfHost` is called before the scheduler is started, the message will not be displayed in SystemView. In main(), `assert_param` is called before the scheduler is started, so the message is not displayed.



You will learn more about using assertions and how to configure them in *Chapter 16, Troubleshooting Tips and Next Steps*.

Now that we have created some tasks, let's get the scheduler started and the code running on our hardware, and watch some lights blink!

Starting the scheduler

With all of the options we have for creating tasks, you might be thinking that starting the scheduler would be a complex affair. You'll be pleasantly surprised at how easy it is:

```
vTaskStartScheduler();
```

Yep, just one line of code and no input parameters!

The v in front of the function name indicates it returns void. In reality, this function never returns – unless there is a problem. It is the point where `vTaskStartScheduler()` is called that **the program transitions from a traditional single super-loop to a multi-tasking RTOS**.

For FreeRTOS APIs, their name typically has a **lower-case prefix**, as in `vTaskStartScheduler` and `xTaskCreate`. The prefix typically indicates the return value's data type. The prefix "x" is typically used for integer types. For `xTaskCreate`, the returned value's data type is `BaseType_t`, and for our MCU, it resolves to type `long`. For the API names, the prefixes are specified on the FreeRTOS website, at the page *Coding Standard, Testing and Style Guide* (`https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html`).

Now that we've learned how to create tasks and get the scheduler up and running, the last detail to cover in this example is how to go about deleting a task.

Deleting tasks

In some cases, it is advantageous to have a task run, and after it has accomplished everything needed, then remove it from the system. For example, in some systems with fairly involved startup routines, it might be advantageous to run some of the late initialization code inside a task. In this case, the initialization code would run, but there is no need for an infinite loop. If the task is kept around, it will still have its stack and **Task Control Block** (**TCB**) wasting FreeRTOS heap space. Deleting the task will free the task's stack and TCB, making the RAM available for reuse. (In general, though, the critical initialization code should be run long before the scheduler starts.)



Many embedded programs will never delete tasks. There are many applications that do not require freeing up resources or dynamically allocating memory.

The task deletes itself

The simplest way to delete a task after it has finished doing useful work is to call **vTaskDelete()** with a `NULL` argument, from within the task. This is done by `GreenTask`:

```
void GreenTask(void *argument)
{
    . . .

    SEGGER_SYSVIEW_PrintfHost("GreenTask started");

    GreenLed.On();
    vTaskDelay(1500 / portTICK_PERIOD_MS);
    GreenLed.Off();

    SEGGER_SYSVIEW_PrintfHost("GreenTask is deleting itself");
    // A task can delete itself by passing NULL to vTaskDelete
    vTaskDelete(NULL);

    // The task never gets to here
    GreenLed.On();
}
```

Deleting the task will immediately terminate the task code. The memory on the FreeRTOS heap associated with the TCB and task-stack will be freed when the Idle-task runs. So, after running `vTaskDelete`, it is essential that the Idle task is not starved of processing time. (The Idle task is described later, in the present chapter.)

In this example, the green LED will turn on for 1.5 seconds and then shut off. As noted in the code, the instructions after `vTaskDelete()` will never be reached.

Deleting a task from another task

In our example-program, `RedTask` deletes `BlueTask` by calling `vTaskDelete`, and passing `BlueTask`'s **task handle**, as shown below. The task handle is in the global variable `blueTaskHandle`. When `xTaskCreate` is called to create `BlueTask`, the task handle's address (`&blueTaskHandle`) is passed in, and the task handle is provided there on return.

```
TaskHandle_t blueTaskHandle;
. . .
int main(void)
{
    . . .
    assert_param( xTaskCreate(BlueTask, "BlueTask", STACK_SIZE, NULL,
                              tskIDLE_PRIORITY + 1,
                              &blueTaskHandle) == pdPASS);
    . . .
}

void RedTask( void* argument )
{
    . . .
            vTaskDelete(blueTaskHandle);
    . . .
}
```

The actual code in `main.c` results in the blue LED blinking for about 2 seconds before being deleted by `RedTask`. At that point, the blue LED stops blinking (since the task turning the LED on/off isn't running anymore).

There are a few things to keep in mind before deciding that deleting tasks is desirable:

- The **heap implementation used must support freeing memory**. For example, `vTaskDelete` cannot be used with `Heap_1`, because it does not free memory. `vTaskDelete` can be used with `Heap_4`, which is what our system uses. (Heap types are explained later in this chapter.)
- Like any heap implementation, it is possible for a heavily used heap to become fragmented if different-sized elements are constantly being added and removed.
- In the FreeRTOS configuration file (`FreeRTOSConfig.h`), this setting must be specified:

  ```
  #define INCLUDE_vTaskDelete  1
  ```

That's it! We now have a FreeRTOS application. Let's get everything compiled and program the image onto the dev-board.

Trying out the code

Now that you've learned how to set up a few tasks, let's go through how to get it running on our hardware. You can greatly enhance your intuition of how an RTOS behaves by running the examples, experimenting with breakpoints to observe execution, and sifting through traces in SystemView.

Building and running the code

First, we'll need to build the code. The IDE-project `Chapter_07` contains two different programs, and each has its own **build configuration**. The program we've been presenting is in `Src\main_taskCreation.c`. The other program is in `Src\main_FailedStartup.c`, and we'll look at it later in the chapter.

To build `main_taskCreation.c`, we need to make its build-configuration **Active**:

1. In the IDE, open the workspace for the book's example-programs, and in Project Explorer, right-click on the `Chapter_07` project.
2. Next, click on **Build Configurations** : **Set Active** : **TaskCreationBuild...**, as shown below:



Figure 7.1 : Setting the active build-configuration

`main_taskCreation.c` can now be built by right-clicking on the project, then clicking on **Build Project**. There should be no errors or warnings. The build directory is `\TaskCreationBuild`, and the executable file is `Chapter_07_TaskCreationBuild.elf`. If the executable file is not visible, right-click on the project and select **Refresh**.

Next we'll use Ozone to load the executable-file on the dev-board, and run it. Create an Ozone project-file (`.jdebug`), as described in the prior chapter. In configuring the file, specify the path to the executable file. Also, uncomment the line shown below, which is needed when FreeRTOS is being used.

```
Project.SetOSPlugin("FreeRTOSPlugin_CM7");
```

Use Ozone to load the executable and start it. `GreenTask` is the highest-priority task, so it will run first. It will spin until SystemView is started, in Record mode.

Experiment with using Ozone to single-step through the program (this was covered in *Chapter 3, Introducing the Development Tools*). SystemView can also be used to watch the tasks run in real time (this was covered in the prior chapter). For this program, it should be possible to run Ozone in debug-mode, and SystemView, at the same time.



If SystemView reports overflow, try running it without Ozone in debug mode.

Additional Ozone features

There are some additional Ozone features that are worth trying here. If you'd like to set a **breakpoint** in a source file that is not displayed, the **Source Files window** can be used to display the file. (The window can be opened via **View** : **Source Files**.) The window lists the executable's source files, as shown below. A file can be displayed by double-clicking it. Similarly, the **Functions window** lists all of the executable's functions (**View** : **Functions**). A function's source file can be displayed by double-clicking on the function.

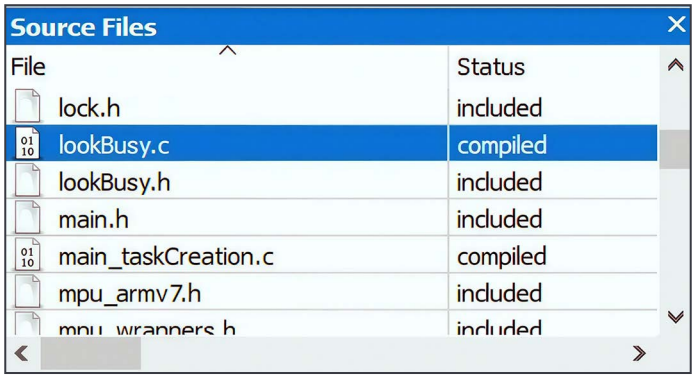


Figure 7.2 : Ozone's Source Files window

When using Ozone with J-Link, a limited number of **hardware breakpoints** can be set. The limit is set by the processor, and with our Arm Cortex M7 the limit is 8. Setting a breakpoint at a **source-code instruction** often results in hardware breakpoints being set for more than one of its **machine instructions**. One way to conserve hardware breakpoints is to set source-code breakpoints on instructions for which only one hardware breakpoint is set. The figure below shows two source-code breakpoints, in `RedTask`. The first source-code breakpoint (**line #160**) uses one hardware breakpoint, and the second (**line #161**) uses two. (In your Ozone source-code view, the assembly language and the hardware breakpoints may look different.)

In addition, our dev-board's J-Link supports unlimited **software breakpoints.** When setting a breakpoint, if no more hardware breakpoints are available, then Ozone will prompt for use of a software breakpoint. A pop-up window states that use of software breakpoints is permitted "for evaluation only" (not shown in the following figure). (More information on breakpoints can be found in the user manuals for J-Link and Ozone, and in the *Arm Cortex-M7 Devices User Guide*.)

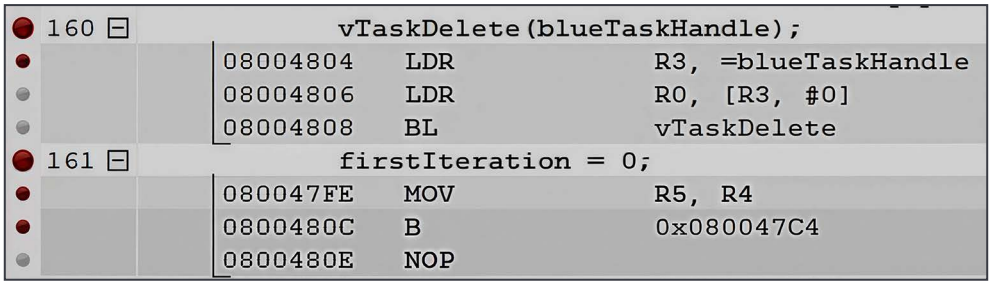


Figure 7.3 : Source-code breakpoints, and their resultant hardware breakpoints

Ozone's breakpoints are listed in the **Break & Tracepoints window** (click on **View : Break & Tracepoints**). This window is especially useful for managing multiple breakpoints, and it has a feature for clearing all breakpoints (**Clear All**). The window is shown below, with its pop-up window that is activated by right-click.

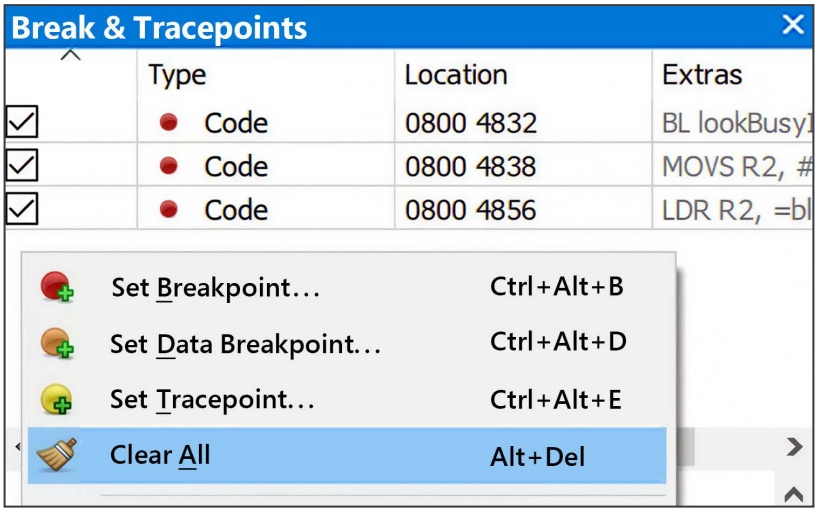


Figure 7.4 : Break & Tracepoints window

Using SystemView to analyze task execution

These SystemView windows provide a bird's-eye view of what's going on when the program runs:

#	Time	Context	Event	Detail
14	0.000 405 009	Idle	ISR Exit	Returns to Idle
15	0.000 916 903	Idle	Log	GreenTask started
16	0.000 923 069	Idle	vTaskDelay	xTicksToDelay=1500
17	0.000 926 889	Idle	Task Block	GreenTask, Delayed
18	0.000 935 810	BlueTask	Task Run	Runs for 28.880 us
19	0.000 953 324	BlueTask	Log	BlueTask is starting a loop iteration
20	0.000 960 699	BlueTask	vTaskDelay	xTicksToDelay=200
21	0.000 964 690	BlueTask	Task Block	Delayed
22	0.000 973 051	RedTask	Task Run	Runs for 1 006.822 ms
23	0.000 990 528	RedTask	Log	RedTask is starting a loop iteration
24	0.000 996 917	SysTick	ISR Enter	Runs for 3.361 us
25	0.001 000 278	SysTick	ISR Exit	Returns to RedTask
26	0.001 995 389	SysTick	ISR Enter	Runs for 3.176 us
27	0.001 998 565	SysTick	ISR Exit	Returns to RedTask

Figure 7.5 : SystemView Event List window

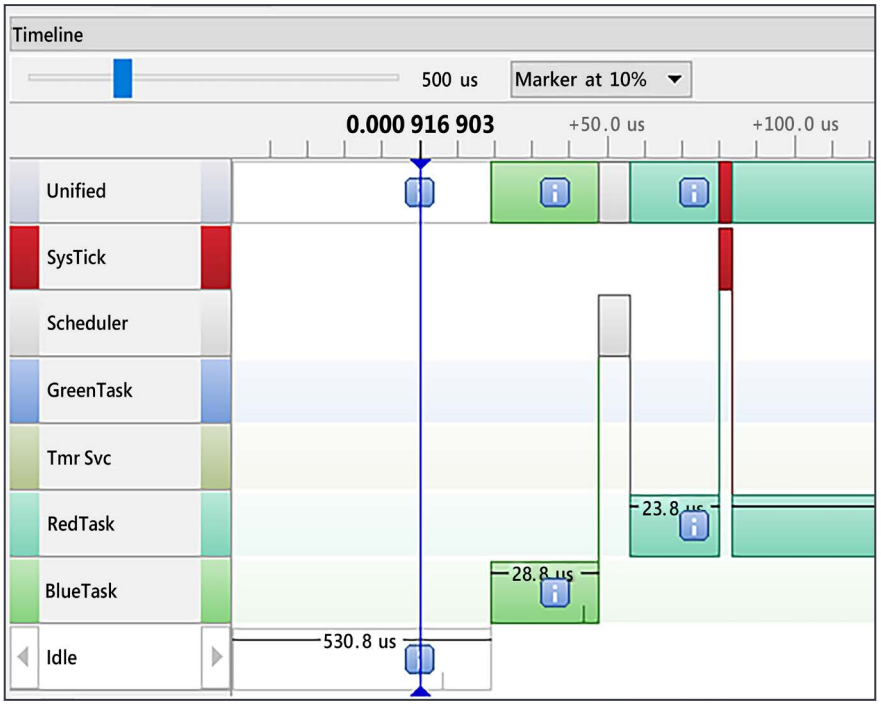


Figure 7.6 : SystemView Timeline window

The above figures show the program's execution, just after the `while` loop in `GreenTask`. The `while` loop was spinning, and it exited when SystemView's Record mode was started.

In the Timeline, the blue marker is at event #15 (shown in the Event List). At that point, `GreenTask` is running, but it is not shown in the Timeline. The apparent reason is that **the Timeline only shows tasks that started after SystemView's Record mode** is started. So, events #15 to 17 incorrectly indicate the `Idle` task is running, though the `GreenTask` is actually running then. As shown, at event #15, `GreenTask` generates the message, **GreenTask started**.

After the Timeline marker, the events illustrate the scheduler's role in switching tasks in and out:

- At event #16, `GreenTask` enters a delay, and it will last for 1,500 ticks (ms).
- At event #17, `GreenTask` calls the scheduler to change `GreenTask` from the Running state to the Blocked state.
- The scheduler runs between events #17 and #18. `BlueTask` and `RedTask` are tied as the highest-priority tasks in the Ready state. Using round-robin scheduling, `BlueTask` is chosen to run next. So the scheduler switches-in `BlueTask`, and puts it in the Running state. The Timeline doesn't show the scheduler running here. Presumably, this is because `GreenTask` started before SystemView's Record mode started.
- At events #18-20, `BlueTask` runs until it enters a delay, and the delay will last for 200 ticks (ms).
- At event #21, `BlueTask` changes from the Running state to the Blocked state.
- The scheduler runs between events #21 and #22, and the scheduler is also shown in the Timeline. `RedTask` is the highest-priority task in the Ready state. So, the scheduler switches-in `RedTask`, and puts it in the Running state.
- At event #22, `RedTask` starts running.
- Shortly after `RedTask` starts, it generates the message **RedTask is starting a loop iteration**, at event #23. `RedTask` then starts its loop, which spins for 1 second of processor time. That loop will get interrupted when `BlueTask`'s delay ends, which is approximately 200 ticks after `RedTask` starts (not shown in the Timeline).
- Events #24-25 show the SysTick ISR running. It is also shown in the Timeline. The SysTick ISR has interrupted `RedTask`. Scheduler code gets called from the ISR, and it determines a context switch is not needed. So, the ISR returns, and `RedTask` resumes.

The Terminal window lists the tasks' messages, as shown below. There, we can see the program's progress at a higher level. The timestamps are formatted such that 0 is when Record mode started.

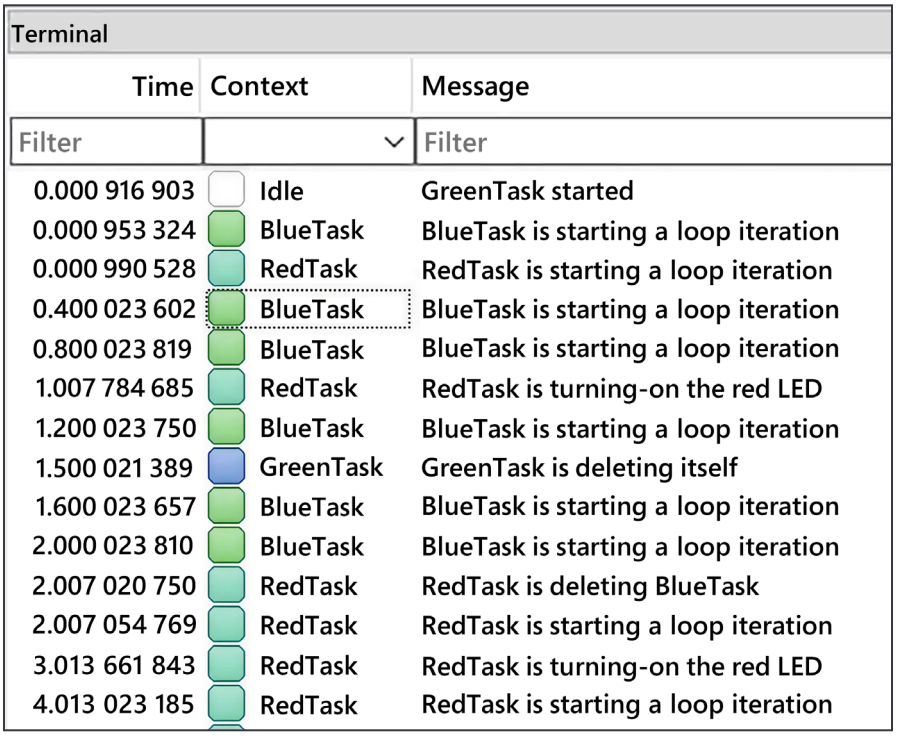


Figure 7.7 : SystemView Terminal window

- We can see that `GreenTask` deleted itself about 1.5 seconds after Record mode started.
- `BlueTask` ran 5 full loop-iterations before it was deleted, which occurred about 2 seconds after `BlueTask` first started.
- `RedTask` is then the only remaining task, and its iteration period is about 2 seconds.

Thus far, if you've just read along with the program description, and not run the program on hardware, you should *seriously* consider getting the dev-board, so that you can run the examples on actual hardware, and use the actual development tools. All of the examples in this book are full-blown projects, ready to go! And, mastering FreeRTOS requires working with it. *Appendix C* shows how to create an IDE-project with FreeRTOS, which you can use to create your own programs. *Appendix C* also shows how you can copy an example-program's IDE-project, to modify it and experiment with it.

Having learned more about creating and deleting tasks, we'll next look at the scheduler.

Understanding the scheduler

This section presents the **scheduler** in more depth, including: the four states a task can be in, the Idle task, how the scheduler is run, and the scheduler architecture.

Understanding FreeRTOS task states

As explained in *Chapter 6, Understanding RTOS Tasks*, all of the context switching between tasks happens *in the background*, which is very convenient for the programmer responsible for implementing tasks. This is because it frees them from adding code into each task that attempts to load-balance the system.

Tasks can be in one of four states: **running**, **ready**, **blocked,** and **suspended**. The **state transitions** are modeled using a **state machine**, as shown below. Each transition is the result of either a FreeRTOS API call made by your code, or an action taken by the scheduler when it is called from the SysTick ISR.

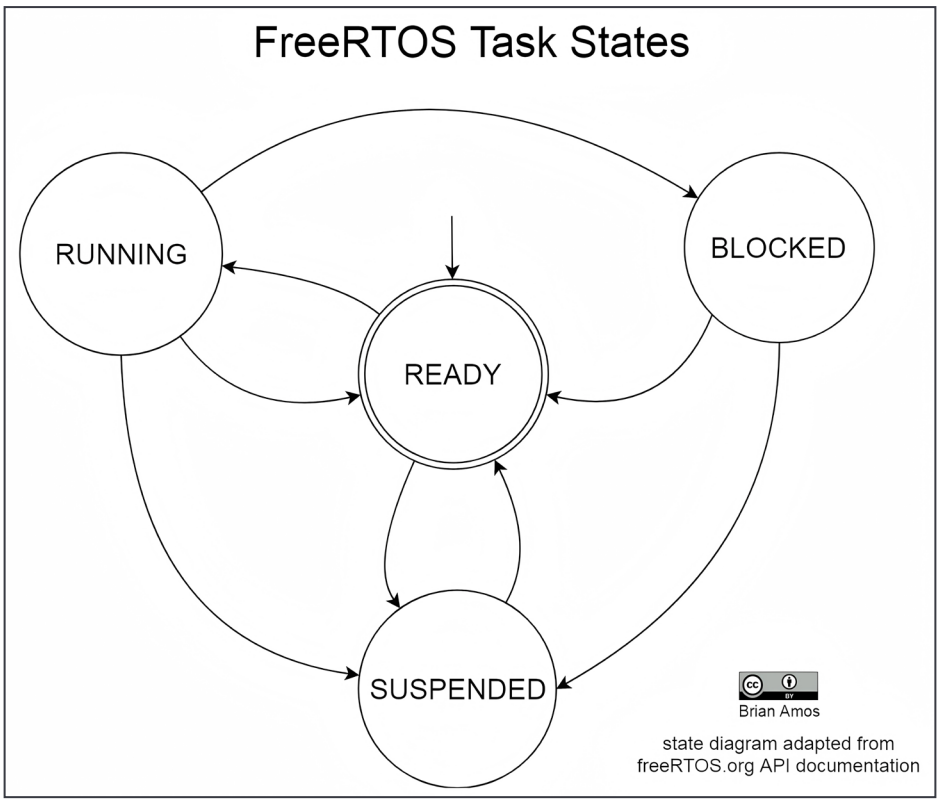


Figure 7.8 : FreeRTOS Task States

Let's look at the states one by one.

Running

Only one task can be in the **Running state** at any one time, on a single core processor. When a task enters the Running state, the task that was executing will have entered some other state. A task in the **Running** state is performing work; it is the only task that is in context. Typically, it will run until either: a) it calls a FreeRTOS API that moves it to the **Blocked** state, or b) it gets switched out of context by the scheduler due to a Blocked task becoming Ready, which has higher-priority (or a time-sliced task of equal priority). For example, `vTaskDelay` causes a task to move from the **Running** state to the **Blocked state.**

Ready

Tasks that are sitting in the **Ready state** are simply waiting for the scheduler to give them processor context so they can run. For example, if Task A has gone into the **Blocked** state due to a call to `vTaskDelay`, then when the delay expires, Task A will move into the **Ready** state. The scheduler will then evaluate whether Task A is the highest-priority task ready to run in the system. If so, it will be given processor context and be changed to the **Running** state. Note that tasks can share the same priority. In that case, the scheduler will switch them between **Ready** and **Running** by using a round-robin scheduling scheme (see *Chapter 6, Understanding RTOS Tasks*, for an example of this).



Switching between tasks in a round-robin scheduling scheme is the default behavior of FreeRTOS. This behavior can be turned off by setting `configUSE_TIME_SLICING` to `0`.

Blocked

A **Blocked task** is a task that is waiting for something. There are two ways for a task to move out of the **Blocked** state: either an event will trigger a transition from **Blocked** to **Ready** for the task, or a timeout will occur, placing the task in the **Ready** state.

In upcoming chapters, we'll learn about FreeRTOS APIs that wait for an event to occur. For instance, events related to a FreeRTOS **queue**. A queue is a circular buffer with a fixed number of elements, each of which is the same size, e.g., a queue with ten `uint8_t` elements. A queue is created by calling **xQueueCreate**.

Elements are added to, and removed from, a queue in first-in-first-out (FIFO) order. **xQueueReceive** removes an element from the queue, and it provides a copy of the element. If the queue is empty, the task will block until an element is available to return. Conversely, **xQueueSend** puts an element in the queue. If the queue is full, the task will block until the element can be put in the queue. Both `xQueueReceive` and `xQueueSend` have a parameter for specifying the maximum time the call can block. The function's return value indicates if it timed-out or was successful.

This is a very important feature of an RTOS: *blocking API-calls are time-bound*. That is, a task will block while waiting for an event only for as long as the programmer specifies it can be blocked. For example, *an attempt to receive a queue element will fail if an element isn't available within the specified amount of time*. The use of blocking API-calls that are time-bound is an important distinction between RTOS firmware programming and general-purpose application programming. The latter typically does not have strict timing requirements, so time-bound blocking API-calls are not used as much.

While a task is in the **Blocked** state, it doesn't consume any processor time. When a task is transitioned out of the **Blocked** state by the scheduler, it will be moved to the **Ready** state, allowing the calling task to run when it becomes the highest-priority Ready task.

Suspended

The **Suspended state** is a bit of a special case since it requires explicit FreeRTOS API calls to enter and exit. Once a task enters the **Suspended** state (via the `vTaskSuspend()` API call), it is effectively ignored by the scheduler until the `vTaskResume()` API call is made. This state causes the scheduler to effectively ignore the task until it is moved into the **Ready** state by an explicit API call. Just like the **Blocked** state, the **Suspended** state will not consume any processor time.



Most applications do not use the Suspended state.

The Idle task

There must always be at least one task that can enter the Running state. To ensure this, an **Idle task** is created by the scheduler when `vTaskStartScheduler()` is called. The Idle task is little more than a spin loop, and it is always able to run.

The Idle task has the lowest possible priority (priority zero). This ensures that it never prevents a higher-priority task from entering the Running state. By running at the lowest priority, the Idle task will be moved out of the Running state whenever a higher-priority task enters the Ready state. Conversely, when none of the other tasks are in the Ready state, the Idle task will run.

In our example-program, the Idle task consumes about half of the CPU time. After `GreenTask` and `BlueTask` are deleted, the only remaining application task is `RedTask`. In `RedTask`'s infinite loop, each iteration has two 500-ms delays (shown below).

```
    while(1)
    {
        SEGGER_SYSVIEW_PrintfHost("RedTask is starting a loop iteration");
        // Spin for 1 second of processor time
        for (i=0;i<1000;i++){
            lookBusy(iterationsPerMilliSecond);
        }

        SEGGER_SYSVIEW_PrintfHost("RedTask is turning-on the red LED");
        RedLed.On();
        vTaskDelay(500/ portTICK_PERIOD_MS);
        RedLed.Off();
        vTaskDelay(500/ portTICK_PERIOD_MS);
        . . .
    }
```

The Idle task is run during those delays, since there are no other tasks to run, as shown below:

- At event #6114, `RedTask` runs `vTaskDelay`.
- At event #6115, `RedTask` calls the scheduler to change `RedTask` from the Running state to the Blocked state.
- The scheduler runs between events #6115 and #6116, and it is also shown in the Timeline. The Idle task is the only task in the Ready state. So, the scheduler switches-in the Idle task, and puts it in the Running state.

- At event #6116, the Idle task starts running.

6112	3.013 998 264	SysTick	ISR Exit	Returns to RedTask
6113	3.014 656 051	RedTask	Log	RedTask is turning-on the red LED
6114	3.014 663 208	RedTask	vTaskDelay	xTicksToDelay=500
6115	3.014 667 083	RedTask	Task Block	Delayed
6116	3.014 675 227	Idle	System Idle	
6117	3.014 995 690	SysTick	ISR Enter	Runs for 3.921 us
6118	3.014 999 611	SysTick	ISR Exit	Returns to Idle
6119	3.015 995 444	SysTick	ISR Enter	Runs for 3.139 us

Figure 7.9 : SystemView Event List window

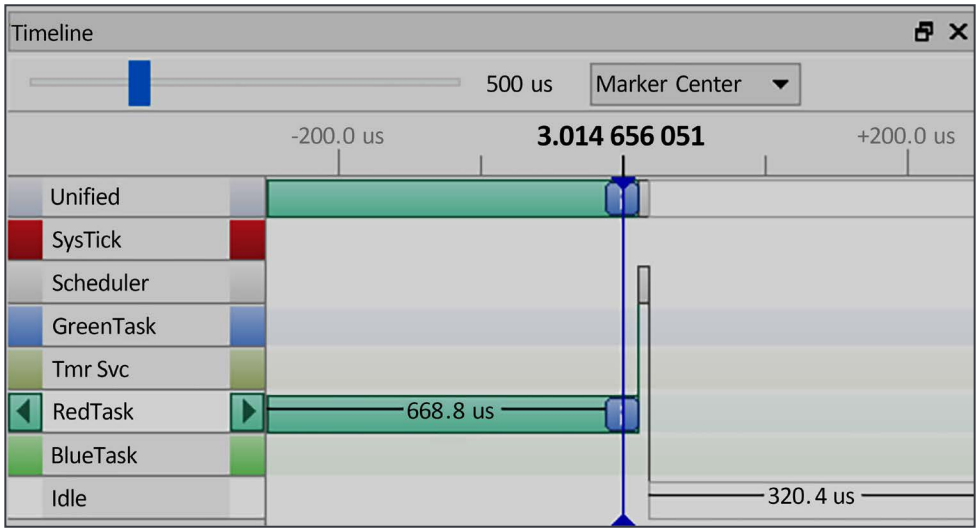


Figure 7.10 : SystemView Timeline window

In the example-program, three tasks are created, and their priorities are calculated relative to the Idle task's priority, e.g., `tskIDLE_PRIORITY + 2` (shown below). This ensures the tasks' priorities are higher than the Idle task's.

```
if ( xTaskCreate(GreenTask, "GreenTask", STACK_SIZE, NULL,
                 tskIDLE_PRIORITY + 2, NULL ) != pdPASS){ while(1); }

assert_param( xTaskCreate(BlueTask, "BlueTask", STACK_SIZE, NULL,
                          tskIDLE_PRIORITY + 1, &blueTaskHandle) == pdPASS );
```

`tskIDLE_PRIORITY` is a system constant with a value of `0`, defined in `task.h`. The maximum number of priorities is defined by the FreeRTOS configuration setting **configMAX_PRIORITIES**, in `FreeRTOSConfig.h`. It is shown below for our system. The range of available priorities is 0 to (`configMAX_PRIORITIES - 1`).

```
#define configMAX_PRIORITIES   ( 56 )
```

A system's spare processing capacity can be calculated based on the amount of processing time allocated to the Idle task. In our example-program, once `RedTask` is the only remaining application task, the Idle task runs about 50% of the time.

An application's tasks can also use the Idle-task priority. In this case, how the Idle task is interrupted depends on the FreeRTOS setting **configIDLE_SHOULD_YIELD**. If `configIDLE_SHOULD_YIELD` is set to `0`, then the Idle task will stay in the Running state for its current time slice (i.e., the SysTick period), unless the Idle task is pre-empted by a higher-priority task. If `configIDLE_SHOULD_YIELD` is set to 1, then the Idle task can yield (give up the rest of its current time slice) on an iteration of its loop. On each iteration, the Idle task will yield if there is another task that has the Idle priority and that is in the Ready state. `configIDLE_SHOULD_YIELD` is specified in `FreeRTOSConfig.h`. If omitted, the default value is 1, which is what our system uses.

How and when the scheduler is run

Now that we understand the various task states and how they interact with different parts of the RTOS, we can learn more about how the scheduler works, and in particular, how and when it is run. In a nutshell, the scheduler is run at every **SysTick interrupt**, and between ticks, the scheduler is run by **FreeRTOS APIs** that change the tasks' scheduling.

The **SysTick ISR calls the scheduler**. The scheduler checks if any timeouts have just occurred for Blocked tasks. An example is a task that is Blocked after calling `vTaskDelay`, and the delay has just ended. Timeouts can also occur for tasks that are Blocked and waiting for an event, but the event hasn't occurred by the specified timeout. This was described earlier for `xQueueReceive` and `xQueueSend`. The scheduler changes these Blocked tasks to the Ready state. And, the scheduler performs a context switch if the highest-priority Ready task has higher priority than the current task. Otherwise, the SysTick ISR returns, and the current task resumes.

The figures below illustrate this process. They show the example-program, after `GreenTask` and `BlueTask` were deleted. The Timeline shows the Idle task is running. This is because `RedTask` ran `vTaskDelay`, and entered the Blocked state.

- At event #8117, the SysTick ISR is run. In the Timeline, the blue marker is at that event.

- The SysTick ISR calls the scheduler. At this point, `RedTask`'s delay has just ended.
- At event #8118, the scheduler moves `RedTask` from the Blocked state to the Ready state.
- The scheduler then does a context switch to the highest-priority task in the Ready state, which is `RedTask`
- At event #8119, `RedTask` starts to run:



Figure 7.11 : The Event List, showing the SysTick ISR running the scheduler

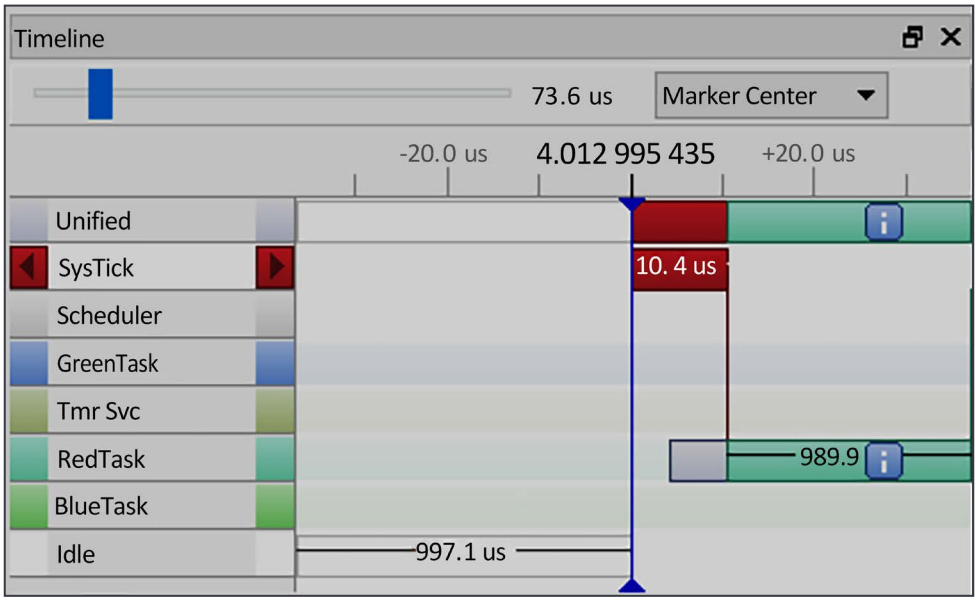


Figure 7.12 : The Timeline, for the Event List

Between ticks, **the scheduler can be run by FreeRTOS APIs**. If a FreeRTOS API is called, and the API affects task-scheduling, then the API will call the scheduler. For example, when `vTaskDelay` is called, it will call the scheduler to a) change the current task to the Blocked state, and b) change the context to the highest-priority Ready task. In general, when a FreeRTOS API needs to change a task's state, the API will call a scheduler function to make the change. And, the API will determine if a context switch is needed, due to the state change. If so, the API will call a scheduler function to perform the context switch.

There are some APIs that do not change a task's state, but which can require a context switch. Examples include `xTaskDelete` (deletes a task) and `vTaskPrioritySet` (changes a task's priority). For such an API, it will determine if a context switch is needed, and if so, it will call a scheduler function to perform the context switch.

The scheduler process for `vTaskDelay` was described in detail earlier, for the example-program.

The scheduler architecture

In using FreeRTOS, it's helpful to have a basic understanding of the scheduler's architecture. Descriptions of FreeRTOS tend to give the impression that "the scheduler" is a monolithic and encapsulated component. Instead, the FreeRTOS scheduler is a collection of code that does "scheduling stuff." And, this code permeates the kernel and the FreeRTOS APIs. The kernel's scheduler-code includes internal C-functions and macros that provide scheduling services, such as changing a task's state, or performing a context switch. The scheduler-code also includes code in FreeRTOS APIs that carry-out the scheduler processing needed for the API, and examples include determining if a context switch is needed, and making calls to internal scheduler-functions.

The scheduler-code used in the SysTick ISR and in FreeRTOS APIs will be shown, as examples. First, two of our processor's operating modes need to be described: **handler mode** and **thread mode**. Handler mode is used when executing an exception handler such as an ISR. Thread mode is used when executing normal application code. Our discussion of FreeRTOS APIs is limited to APIs called from thread mode. There are FreeRTOS APIs that can be called from an ISR, and they are presented later in the book.

The figure above shows the SysTick ISR running, and it was stated that the ISR "calls the scheduler." Internally, the ISR calls the scheduler function `xPortSysTickHandler`. That function runs in handler mode, and ultimately returns to the ISR. The function performs scheduling activities, such as calling a scheduler-function that increments the tick counter and moving appropriate tasks in the Blocked state to the Ready state. `xPortSysTickHandler` then determines if a context switch is needed, and if so, it generates a `PendSV` interrupt. The `PendSV` ISR calls a scheduler-function to carry-out the context switch.

Note that in the SystemView Timeline (above), the SysTick ISR is shown, but the Timeline doesn't show the scheduler running from the ISR. In contrast, when a FreeRTOS API "calls the scheduler," the SystemView Timeline shows the scheduler running, in the **Scheduler** row. This can be seen in the figures below. The figures are from the earlier example of the Idle task being run, after `RedTask` called `vTaskDelay`.

6112	3.013 998 264	SysTick	ISR Exit	Returns to RedTask
6113	3.014 656 051	RedTask	Log	RedTask is turning-on the red LED
6114	3.014 663 208	RedTask	vTaskDelay	xTicksToDelay=500
6115	3.014 667 083	RedTask	Task Block	Delayed
6116	3.014 675 227	Idle	System Idle	
6117	3.014 995 690	SysTick	ISR Enter	Runs for 3.921 us
6118	3.014 999 611	SysTick	ISR Exit	Returns to Idle
6119	3.015 995 444	SysTick	ISR Enter	Runs for 3.139 us

Figure 7.13 : Event List, showing the scheduler being called from vTaskDelay

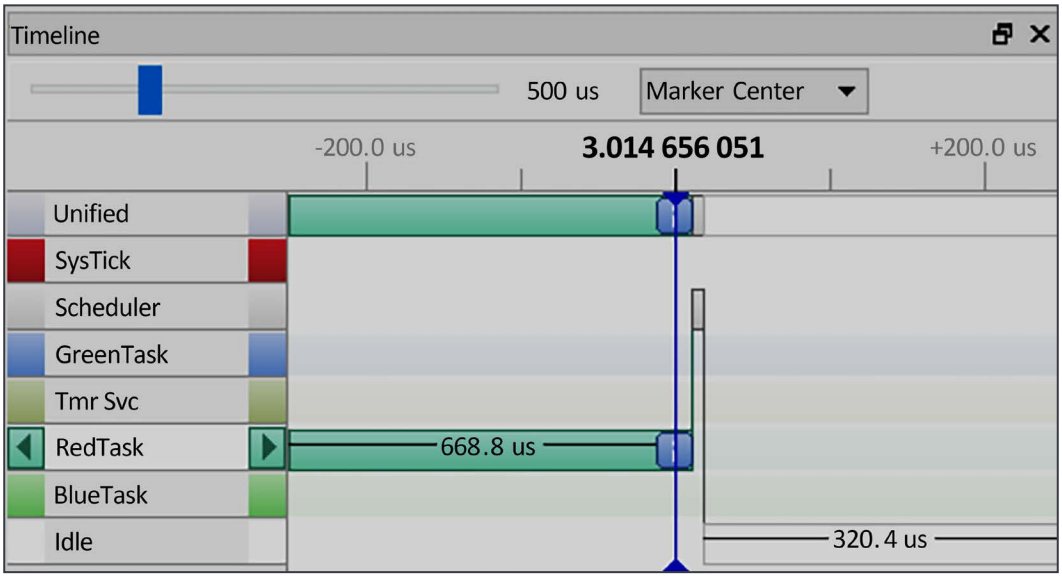


Figure 7.14 : Timeline, for the Event List

These figures illustrate the scheduler-code used in `vTaskDelay`. When `vTaskDelay` "calls the scheduler," it doesn't call a single scheduler-function, like the SysTick ISR does. Instead, `vTaskDelay` calls a scheduler-function that moves the current task from the Running state to the Blocked state. The Event List shows this at event #6115 (though it's shown as running under `RedTask`, not the Scheduler). `vTaskDelay` then does some other scheduler stuff, and eventually calls a C macro that performs the context switch. The macro generates a `PendSV` interrupt, which was described earlier. This scheduler processing is shown in the SystemView Timeline, in the *Scheduler* row, and it is between events #6115 and #6116.

The description here of the SysTick ISR's scheduler-process is from the FreeRTOS web page *The RTOS Tick* (`https://www.freertos.org/implementation/a00011.html`), and from the FreeRTOS source-code. The description of `vTaskDelay`'s scheduler process is from the source code. The other FreeRTOS APIs that affect scheduling appear to have similar scheduler code.

Task implementation and troubleshooting

This section presents several topics related to task implementation, and to troubleshooting task startup problems.

Task memory allocation

One of the things we glossed over previously was why a call to `xTaskCreate()` can *fail*. That's an excellent question – let's find out!

One of the parameters for `xTaskCreate()` defines the task's stack size. But where does the RAM for this stack come from? There are two options – **dynamic allocation of memory** and **static allocation of memory**.

Dynamic allocation of memory is implemented with a heap. FreeRTOS contains several different heap implementations. FreeRTOS heaps will be explained later. For now, it is sufficient to assume a heap is available.

Static allocation of memory permanently reserves RAM for data, for the life of the program. Let's see what each approach looks like.

Dynamically allocated tasks

Our example-program has two calls to `xTaskCreate`. This API uses the heap to dynamically allocate memory for the task's stack:

```
#define STACK_SIZE 128
. . .
xTaskCreate(GreenTask, "GreenTask", STACK_SIZE, NULL,
            tskIDLE_PRIORITY + 2, NULL )
```

`xTaskCreate()` is the simpler of the two APIs for task creation. It will use memory from the FreeRTOS heap for the task's stack and for the **TCB**. As described earlier, the amount of memory is specified by `STACK_SIZE`, whose units are words. With our processor, words are 4 bytes. So, 512 bytes are used for the task's stack and TCB (4 * 128 bytes).

Statically allocated tasks

Tasks that are created without using the FreeRTOS heap require the programmer to perform allocation for the task's stack and TCB before creating the task. The static version of task creation is `xTaskCreateStatic()`.

The FreeRTOS prototype for `xTaskCreateStatic()` is as follows:

```
TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
                                const char * const pcName,
                                const uint32_t ulStackDepth,
                                void * const pvParameters,
                                UBaseType_t uxPriority,
                                StackType_t * const puxStackBuffer,
                                StaticTask_t * const pxTaskBuffer );
```

Let's take a look at how this is used in our example, which creates a task with a statically allocated stack:

```
#define STACK_SIZE 128
static StackType_t RedTaskStack[STACK_SIZE];
static StaticTask_t RedTaskTCB;
xTaskCreateStatic( RedTask, "RedTask",
                   STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, RedTaskStack,
                   &RedTaskTCB);
```

Unlike `xTaskCreate()`, `xTaskCreateStatic()` is guaranteed to always create the task, provided `puxStackBuffer` and `pxTaskBuffer` are not `NULL`. As long as your toolchain's linker can find space in RAM to store the variables, the task will be created successfully.



To make use of the preceding code, `configSUPPORT_STATIC_ALLOCATION` must be set to `1` in `FreeRTOSConfig.h`.

Memory-protected task creation

Tasks can also be created in a memory-protected environment, which guarantees a task only accesses memory specifically assigned to it. FreeRTOS provides this feature on MCUs with a **memory protection unit (MPU)**, like our STM32F767 MCU.

An MPU continuously monitors memory access at a hardware level to make absolutely certain that only legal memory accesses are occurring; otherwise, an interrupt is raised and immediate action can be taken. This allows many common errors to be immediately detected, which might otherwise go unnoticed for a period of time.

More information on using an MPU with FreeRTOS is in *Chapter 14, FreeRTOS Memory Management*.

Task creation roundup

Since there are a few different ways of creating tasks, you might be wondering which one should be used. All implementations have their strengths and weaknesses, and it really does depend on several factors. The following table shows a summary of the three ways of creating tasks, with their relative strengths represented by arrows: ⇑ for better, ⇓ for worse, and ⇔ for neutral:

Characteristic	**Heap**	**MPU Heap**	**Static**
Ease of use	⇑	⇓	⇔
Flexibility	⇑	⇓	⇔
Safety	⇓	⇑	⇔
Regulatory Compliance	⇓	⇑	⇑
Linker indication of memory availability	⇓	⇓	⇑

As we can see, there is no clear-cut answer as to which system to use. However, if your MCU doesn't have an MPU on board, there won't be an option to use the MPU variant.

The FreeRTOS heap-based approach is the easiest to code, as well as the most flexible of the three choices. This flexibility comes from the fact that the tasks' stack space is freed when the task is deleted. Statically created tasks are the next easiest, with only an additional two lines required to specify the TCB and the task stack. They aren't as flexible since there is no way to free memory defined by static allocation. Static creation can also be more desirable in some regulatory environments that forbid the use of the heap altogether, although in most cases, the FreeRTOS heap-based approach is acceptable – especially heap implementations 1, 2, and 3.



What is a heap implementation? Don't worry about it yet. We'll learn about heap options for FreeRTOS in detail in *Chapter 14, FreeRTOS Memory Management*.

The MPU variant is the most involved of the three, but it is also the safest since the MPU guarantees the task isn't writing outside of its allowed memory.

Using statically defined stacks and TCBs has the advantage that the total program footprint can be analyzed by the linker. This ensures that if a program compiles and fits into the hardware constraints of the MCU, it won't fail to run due to a lack of heap space. With heap-based task creation, it is possible for a program to compile but have a runtime error that causes the entire application not to run. In other cases, the application may run for some time, but then fail due to a lack of heap memory.

Optimizing task states

Having learned about the various task states and how they interact with different parts of the RTOS, we can now learn how to optimize an application so that it makes efficient use of tasks.

Thoughtful optimizations can be made to minimize the time tasks stay in the Running state. Since a task only consumes significant CPU time while in the Running state, it is usually a good idea to minimize the time spent there on legitimate work.

As you'll see, polling for events does work but is usually an unnecessary waste of CPU cycles. If properly balanced with task priorities, the system can be designed to be both responsive to important events, while also minimizing CPU time. There can be a few different reasons for optimizing an application in this way.

Optimizing to reduce CPU time

Often, an RTOS is used because many different activities need to happen almost simultaneously. When a task needs to take action because an event occurs, there are a few ways of monitoring for the event.

Polling is when a value is continuously read in order to capture a transition. An example of this would be waiting for a new ADC reading. A **polled** read might look something like this:

```
uint_fast8_t freshAdcReading = 0;
while(!freshAdcReading)
{
    freshAdcReading = checkAdc();
}
```

While this code *will* detect when a new ADC reading has occurred, it will also cause the task to continually be in the Running state. If this happens to be the highest-priority task in the system, this will **starve** the other tasks of CPU time. This is because there is nothing to force the task to move out of the Running state – it is continually checking for a new value.

To minimize the time a task spends in the Running state (continually polling for a change), we can use the hardware included in the MCU to perform the same check without CPU intervention. For example, ISRs and **direct memory access (DMA)** can both be used to offload some of the work from the CPU onto different hardware peripherals included in the MCU. An ISR can be interfaced with RTOS primitives to notify a task when there is valuable work to be done, thereby eliminating the need for CPU-intensive polling. *Chapter 8, Protecting Data and Synchronizing Tasks*, will cover polling in more detail, as well as multiple efficient alternatives.

Optimizing to increase performance

Sometimes, there are tight timing requirements that need a low amount of jitter. Other times, a peripheral requiring a large amount of throughput may need to be used. While it may be possible to meet these timing requirements by polling inside a high-priority task, it is often more reliable (and more efficient) to implement the necessary functionality inside an ISR. It may also be possible to not involve the processor at all by using DMA. Both of these options prevent tasks from expending worthless CPU cycles on polling loops and allow them to spend more time on useful work.

Because interrupts and DMA can operate completely below the RTOS (not requiring any kernel intervention), they can have a dramatically positive effect on creating a deterministic system. We'll look at how to write these types of drivers in detail, in *Chapter 10, Drivers and ISRs*, and *Chapter 11, More Efficient Drivers and ISRs*.

Optimizing to minimize power consumption

With the prevalence of battery-powered and energy-harvesting applications, programmers have another reason to make sure the system is using as few CPU cycles as possible. Similar ideas are present in creating power-conscious solutions, but instead of maximizing determinism, the focus is often on saving CPU cycles and operating with slower clock rates.

There is an additional feature in FreeRTOS that is available for experimentation in this space – the **tickless Idle task**. This trades timing accuracy for a reduction in how often the kernel runs. Normally, if the kernel was set-up for a 1 ms tick rate (waiting up to every millisecond to check for the next activity), the kernel would wake up and run the code at 1 kHz. In the case of a tickless Idle task, the kernel only wakes up when necessary.

Now that we've covered some starting points on how to improve an already running system, let's turn our attention to something more dire: a system that doesn't start at all!

Troubleshooting task startup problems

So, let's say you're working on a project and things haven't gone as planned. Instead of being rewarded with blinky lights, you're left staring at a very non-blinky piece of hardware. At this stage, it's usually best to get the debugger up and running, rather than making random guesses about what might be wrong and sporadically changing sections of code.

Most often, startup problems in the early stages of development will be caused by not allocating enough space in the FreeRTOS heap. There are typically two symptoms that result from this: task creation fails, or the scheduler returns unexpectedly.

Task creation failed

The problem of failed task-creation is illustrated by the example-program `main_FailedStartup.c`. It's in the IDE-project `Chapter_07`. To build this program, set the project's active build-configuration to **FailedStartupBuild**, as shown below. To run the program in Ozone, the project's `.jdebug` file will need to be updated. There, the function-call `File.Open()` should specify the path to the program's `.elf` file, which is `FailedStartupBuild\Chapter_07_FailedStartupBuild.elf`.



Figure 7.15 : Setting the active build-configuration

When running the program, the code will get *stuck* before `GreenTask` starts (the blue LED is not on, and the other LEDs are off). In Ozone, pausing the program shows it is spinning in **assert_failed** (shown below). (To pause the program, click on **Debug : Halt**.)

```
194  void assert_failed(uint8_t *file, uint32_t line)
195  {
196      BlueLed.On();
197      RedLed.On();
198      GreenLed.On();
199      SEGGER_SYSVIEW_PrintfHost("assert failed. Line: &u.  File:", line);
200      SEEGER_SYSVIEW_PrintfHost((char *) file);
201      while(1);
202  }
```

Figure 7.16 : The program is stuck in assert_failed

Perform the following steps to determine why:

1. Using a debugger, step through the task-creation code until you find the offending task. This is easy to do because all of our attempts to create tasks will only progress if the tasks were successfully created.

2. In this case, you'll see that `xTaskCreate` doesn't return `pdPASS` when creating `BlueTask`. The following code is requesting a 50 KB stack (128 * 4 * 100) for `BlueTask`:

```
#define STACK_SIZE 128
. . .
assert_param( xTaskCreate(BlueTask, "BlueTask", STACK_SIZE * 100,
                          NULL,tskIDLE_PRIORITY + 1,
                          &blueTaskHandle)== pdPASS);
```

3. Alternatively, using the Ozone **Call Stack** window, the failed assertion can be tracked back to creating `BlueTask` on line 69 of `main_FailedStartup.c`. (Your line numbers may be slightly different.) In the figure below, the program was paused.

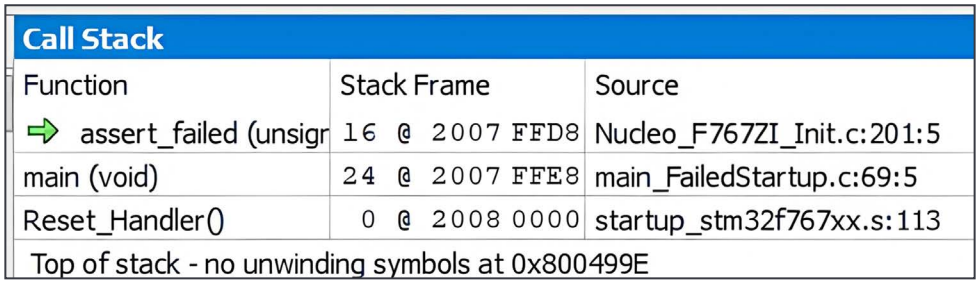


Figure 7.17 : Using the Ozone Call Stack for debugging

4. After determining the cause of failure was a task that failed to be created, it would be prudent to examine `xTaskCreate`'s arguments. The only questionable argument is the **stack-depth**: `STACK_SIZE * 100`.
5. The system's heap size is specified in the FreeRTOS configuration setting **configTOTAL_HEAP_SIZE**, in `FreeRTOSConfig.h`. In our system, the file is in `Inc/FreeRTOSConfig.h`, and the setting is 15 KB, which is much less than the 50 KB requested.

```
#define configTOTAL_HEAP_SIZE ((size_t)15360)
```

6. So, solving the problem would involve decreasing the stack-depth or increasing the system's heap size.



Unlike stack size specifications, which are specified in words (for example, `configMINIMAL_STACK_SIZE`) and passed as arguments to `xTaskCreate`, `configTOTAL_HEAP_SIZE` is specified in bytes.

Care needs to be taken when increasing `configTOTAL_HEAP_SIZE`. See the section *Important notes*, below, on considerations to be made.

Scheduler returns unexpectedly

It is also possible to run into issues with `vStartScheduler` returning, like this:

```
// Start the scheduler. It should not return unless there's a problem.
vTaskStartScheduler();

// If you've wound up here, there is likely an issue with
// over-running the FreeRTOS heap
while(1) {}
```

This is simply another symptom of the same underlying issue – **inadequate heap space**.



If you're reading this section because you *actually have* a program that isn't starting and you're *not* experiencing either of these symptoms, do not worry! There's an entire chapter in the back of this book, just for you. Check out *Chapter 16, Troubleshooting Tips and Next Steps*. It was actually created from real-world problems that were encountered during the creation of the example code in this book.

There are a few more considerations to be made if you have an application that is refusing to start.

Important notes

RAM on MCU-based embedded systems is usually a scarce resource. When increasing the heap space available to FreeRTOS (`configTOTAL_HEAP_SIZE`), you'll be reducing the amount of RAM available to non-RTOS code.

There are several factors to be aware of when considering increasing the heap available to FreeRTOS via `configTOTAL_HEAP_SIZE`:

- System-initialization functions run before the FreeRTOS scheduler is started, and a non-RTOS stack is needed for those functions. If initialization involves deep function-calls, this stack can be sizable..
- ISRs will be making use of the non-RTOS stack as well, but they'll need it for the entire duration of the program.
- Consider using statically allocated tasks instead – it's guaranteed there will be enough RAM when the program runs.



A more in-depth discussion on memory allocation can be found in Chapter 14, FreeRTOS Memory Management.

Summary

In this chapter, we've covered the different ways of creating tasks and how to start the FreeRTOS scheduler. We also covered how the scheduler works, including the use of task states, scheduling priorities, and the Idle task. The two ways that the scheduler is called were described: from the SysTick ISR, and from FreeRTOS APIs that affect task-states and scheduling.

Along the way, we covered some more examples of using Ozone, SystemView, and STM32CubeIDE (or any Eclipse CDT-based IDE). All of this information was used to run a live demo system that tied together all of the RTOS concepts regarding task creation and scheduling, and it showed the mechanics of actually loading and analyzing code running on embedded hardware. There were also some suggestions on how *not* to monitor for events (polling).

In the next chapter, we'll introduce what you *should* be using for event monitoring. Multiple ways of implementing inter-task signaling and synchronization will be covered – all through examples. There's going to be LOTS of code and a bunch of hands-on analysis using the Nucleo board, so we recommend you get one up and running if you haven't already.

8

Protecting Data and Synchronizing Tasks

What do race conditions, corrupt data, and missed real-time deadlines all have in common? Well, for one, they are all mistakes that can be easily made when operations are performed in parallel. These are also mistakes that are avoidable (in part) through using the right techniques.

This chapter covers many of the RTOS primitives that are used to synchronize tasks and protect shared data. The explanations in this chapter contain example-code, and analysis performed using Ozone and SystemView.

First, we will explore the differences between semaphores and mutexes. Next, you will understand how, when, and why to use a semaphore. You'll also learn about race conditions and see how a mutex can avoid such situations. Example code will be provided throughout. The concept of race conditions will be introduced and fixed using a mutex in live code that can be run and analyzed on the Nucleo development board. Finally, FreeRTOS software timers will be introduced, and a discussion of real-world use-cases for them will be provided.

We will cover the following topics in this chapter:

- Running the chapter's example-programs
- Semaphores (used for synchronizing tasks)
- Priority inversion (how not to use semaphores)
- Mutexes (enhanced semaphores)
- Race conditions (when tasks are not synchronized properly)
- Software timers

Running the chapter's example-programs

The chapter's example-programs are in the IDE-project Chapter_08. There are 6 example-programs, and each has it's own build configuration.

To build a particular build-configuration, it must be made the *active* build-configuration:

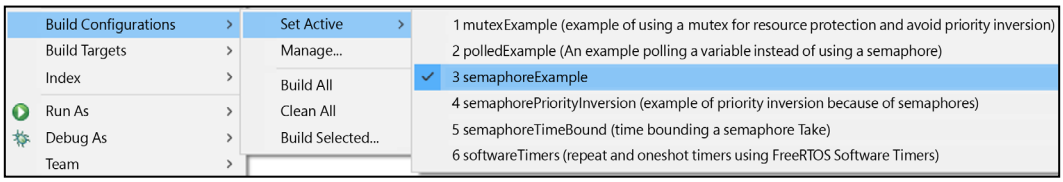


Figure 8.1 : The example-programs' build configurations

Project Explorer identifies files that are not part of the active build-configuration by putting a slash across their thumbnail.

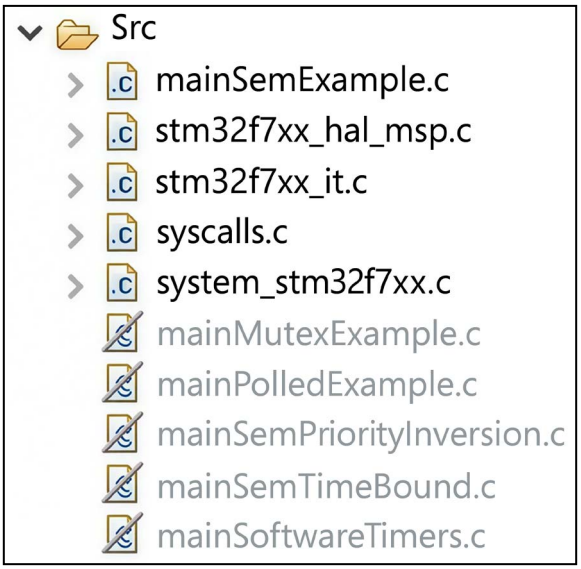


Figure 8.2 : The Project Explorer tags the files not in the active build-configuration

The easiest way to build the examples programs is to build all of them at once:

1. In STM32CubeIDE, open the Console window (**Window | Show View | Console**).

2. In the Console window's toolbar, click on the icon **Open Console**, and select **C/C++ Build Console**.

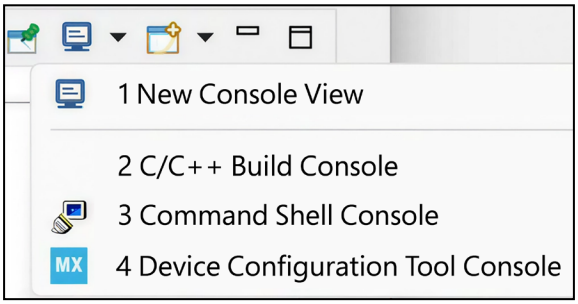


Figure 8.3 : Selecting the C/C++ Build Console

3. Open the "CDT Global Build Console". In the Console window's toolbar, click on the icon **Display Selected Console**, then select **CDT Global Build Console**. This console window will show the build messages for all of the build configurations.

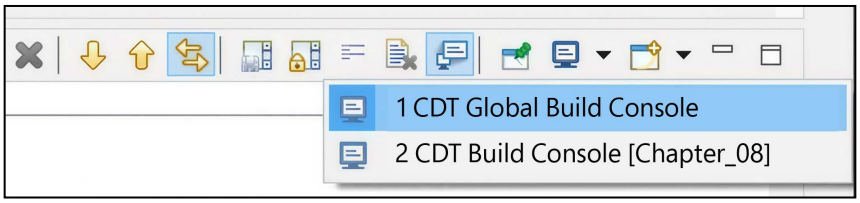


Figure 8.4 : Opening the "CDT Global Build Console"

4. If needed, clear the console window via right-click+**Clear**.
5. Build the example-programs. In the Project Explorer, right-click on **Chapter_08**, and select: **Build Configurations | Build All**.
6. All the examples will be built into their own build-subdirectory, in the Chapter_08 project (this may take a while), e.g., /`mutexExample`, etc. If not visible in Project Explorer, a Refresh may be needed.
7. Confirm the builds worked by searching in the Console window for "Build Finished."

To run the programs under Ozone, create an Ozone project-file (`.jdebug`), as described in the prior chapter. In configuring the project-file, create six calls to `File.Open()`, one for each example-program.

```
File.Open ("<specify full-path for the .elf file to be debugged>");
```

In each call, specify the path to an example-program's `.elf` file, in its build-subdirectory. Before calling Ozone, comment-out all of the calls to `File.Open()`, except for the program to be run.

Next, we'll address the topics related to synchronizing tasks.

Semaphores

We've mentioned several times now that tasks are meant to be programmed so that they're *running in parallel*. This means that, by default, they have no relation to one another in time. No assumptions can be made as to where tasks are in their execution with respect to one another – unless they are explicitly synchronized. **Semaphores** are one mechanism that's used to provide synchronization between tasks.

Semaphores are presented in the following two sections. The first section describes semaphores conceptually. The second section shows how to use FreeRTOS's semaphores, via example-programs.

Understanding semaphores

Semaphores are a very straightforward, but powerful, construct. The word *semaphore* has a Greek origin – the approximate English translation is *sign-bearer*, which is a wonderfully intuitive way to think about them. Semaphores are used to indicate that something has happened – they signal events. Some example use-cases of semaphores are:

- An ISR is finished servicing a peripheral. The ISR may *give* a semaphore to provide tasks with a signal indicating that data is ready for further processing.
- A task has reached a juncture where it needs to wait for other tasks to *catch-up*, before the task can move on. In this case, a semaphore could be used to synchronize all of the tasks.
- Limiting the number of simultaneous users of a restricted resource, e.g., a `struct`.

One of the convenient aspects of using an RTOS is the ubiquitous existence of semaphores. They are included in every implementation of an RTOS because of how basic (and crucial) their functionality is. There are two different types of semaphores to cover: counting semaphores and binary semaphores.

Counting semaphores

Counting semaphores are most often used to manage a shared resource that has limitations on the number of simultaneous users. When the counting semaphore is created, the maximum number of resources is specified, and, the semaphore's counter is typically initialized to that value.

The example normally given for counting semaphores is readers in a database. Well, we're talking about an MCU-based embedded system here, so let's keep our examples relevant. If you're interested in databases, you're probably better off with a general-purpose OS! For our example, let's say you're implementing a socket-based communication driver and your system only has enough memory for a limited number of simultaneous socket connections.

In the following diagram, we have a shared network-resource that can accommodate two sockets being used simultaneously. However, there are three tasks that each have a socket that they'd like to use. A counting semaphore is used to limit the number of sockets that are simultaneously in-use. When the counting semaphore is created, the counter's maximum value is specified as 2, and the counter is initialized to 2.

A task uses a socket by reading from the socket and writing to the socket. Each time a task wants to use the shared-resource (that is, use its socket), it must *take* the semaphore. *Taking* the semaphore decreases the semaphore's count by one. Each time a task is finished with the shared-resource (that is, stops using its socket), the task must *give* back its semaphore, so another task can use its own socket. *Giving* the semaphore increases the semaphore's count by one. If a task happens to *take* a semaphore whose count is 0, the count will remain unchanged, and the task will wait for the semaphore to become available.

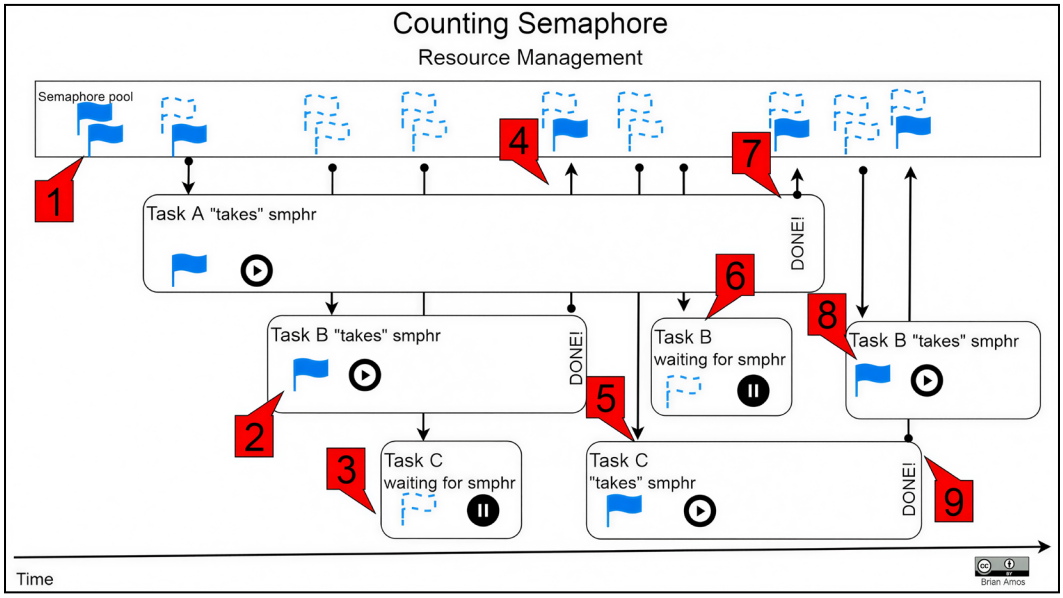


Figure 8.5 : A counting semaphore

The preceding diagram illustrates a shared-resource only capable of servicing two simultaneous tasks (although three tasks in the system need to use the resource). The diagram's numbered steps are explained below. In general, if a task is going to use a socket that is protected by the counting semaphore, the task must first *take* a semaphore from the pool. If no semaphore is available, then the task must wait until a semaphore becomes available.

1. Initially, a semaphore named `smphr` is created. For the semaphore's counter, the maximum count is specified as 2, and its initial value is specified as 2.
2. At point 2, `TaskA` had earlier issued a `take` for `smphr`. It was successful, and `smphr`'s count was decremented to 1. `TaskB` then issues a `take` for `smphr`. It was successful and `smphr`'s count was decremented to 0. Upon getting a semaphore, each task may use its socket, i.e., read from and write to the socket.
3. `TaskC` ran a bit later, and it issues a `take` for `smphr`. However, the count for `smphr` is currently 0, so `TaskC` will need to wait until the count is more than 0.
4. `TaskB` is finished using its socket, so it returns the semaphore. The count for `smphr` is incremented to 1.
5. Now that `smphr` is available, `TaskC` wakes-up, completes its `take`, and `smphr` is decremented to 0. `TaskC` may now use its socket.
6. `TaskB` has another message to send, so it attempts to take `smphr`. However, the count is 0, so `TaskB` will wait until `smphr` becomes available.
7. `TaskA` finishes using its socket and returns `smphr`. The count for `smphr` is now 1.
8. `TaskB` is woken-up, completes its `take`, and `smphr` is decremented to 0. `TaskB` may now start using its socket.
9. `TaskC` finishes using its socket and returns the semaphore. The count for `smphr` is now 1.

The semaphore limits network-use by establishing a convention that the tasks follow: a) a task will not use its socket without first getting the semaphore, and b) when a task is done using its socket, it will return the semaphore. If a task were to use its socket without first getting the semaphore, the semaphore-mechanism would not prevent that.

When a task requests a semaphore, the task can specify the maximum amount of time to wait for the semaphore. If a task fails to acquire the semaphore in time, the task must not access the shared-resource. Instead, the task must take an alternative course of action. This alternative could be any number of actions that can range from a failure so severe that it triggers an emergency shutdown sequence, to something so benign that it is merely mentioned in a log file or pushed to a debug serial-port for analysis later. As a programmer, it is up to you to determine what the appropriate course of action is, which can sometimes prompt difficult discussions with other system developers.

Binary semaphores

Binary semaphores are really just counting semaphores with a maximum count of 1. They are most often used for synchronization. When a task needs to synchronize on an event, it will attempt to *take* a semaphore, and block until the semaphore becomes available, or until the specified timeout has elapsed. Another asynchronous part of the system (either a task or an ISR) will *give* a semaphore. In the following example, `TaskA` only *gives* a semaphore, while `TaskB` only *takes* a semaphore:

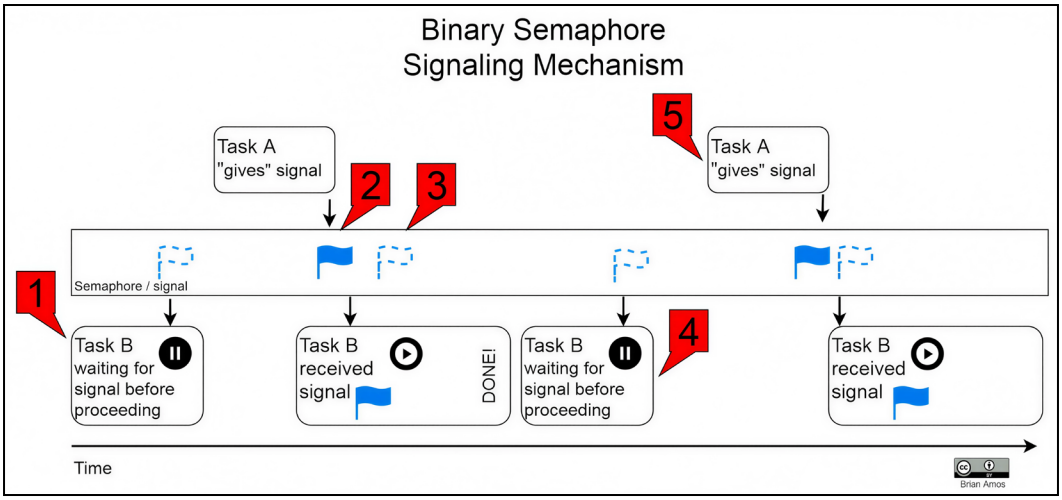


Figure 8.6 : Binary-semaphore signalling-mechanism

`TaskB` is set-up to wait for a signal (semaphore) before proceeding with its duties:

1. Initially, `TaskB` attempts to *take* the semaphore, but it isn't available, so `TaskB` waits for the semaphore to become available.
2. Sometime later, `TaskA` *gives* the signal.
3. `TaskB` is then woken up (by the scheduler; this happens in the background) and `TaskB` now has the semaphore. It will go about the duties required of it until it is finished. Notice, however, that `TaskB` doesn't need to give back the binary semaphore. Instead, it simply waits for it again.
4. `TaskB` attempts to *take* the semaphore, but `TaskB` is blocked again because the semaphore isn't available (just like the first time), so `TaskB` waits until the semaphore is available.
5. The cycle repeats.

In this example, when `TaskB` gets the binary semaphore, `TaskB` does not "give back" the semaphore. If `TaskB` did that, then when `TaskB` next attempts to *take* the semaphore, it would immediately get the semaphore. `TaskB` would immediately proceed, without receiving the go-ahead from `TaskA`. The result would just be a loop running at full speed, rather than having its progress being dependent on a signal from `TaskA`.

The next section presents example-programs that show how to use semaphores.

Using semaphores

This section shows how to use FreeRTOS semaphores. Two example-programs are presented and explained.

Synchronization via binary semaphores

Binary semaphores were just described. An abstract example was shown in the prior figure. That example is implemented in the example-program `mainSemExample.c`. Next, we'll take a look at that code. Then, we'll run it on hardware and blink a few LEDs to see what semaphore-use looks like in the real world.

Understanding the code

The code implements the example that was just shown. Two tasks give and take a semaphore, respectively. First, the semaphore needs to be created, and its **handle** (or pointer) has to be stored so that it can be used by the tasks.

```
// Create storage for a pointer to a semaphore
SemaphoreHandle_t semPtr = NULL;

int main(void)
{
    //... init code removed ...//

    // Create a semaphore using the FreeRTOS Heap
    semPtr = xSemaphoreCreateBinary();

   // Ensure the pointer is valid (semaphore created successfully)
    assert_param(semPtr != NULL);
```



The semaphore pointer, `semPtr`, needs to be placed in a location that is accessible to the functions that access the semaphore. For example, don't declare `semPtr` as a local variable inside a function – it won't be available to other functions, and it will go out of scope as soon as the function returns.

To see what's going on with the source code, *and* see how the system is reacting, we'll associate a different LED with each of the two tasks: `GreenTaskA` and `BlueTaskB`.

`GreenTaskA` will toggle the green LED and *give* the semaphore every five iterations of the task's `while` loop, as shown below. `GreenTaskA` has higher priority than `BlueTaskB`.

```
void GreenTaskA( void* argument )
{
    uint_fast8_t count = 0;
    while(1)
    {
        // Every 5 times through the loop, give the semaphore
        if(++count >= 5)
        {
            count = 0;
            SEGGER_SYSVIEW_PrintfHost("GreenTaskA gives semPtr");
            xSemaphoreGive(semPtr);
        }
        GreenLed.On();
        vTaskDelay(100/portTICK_PERIOD_MS);
        GreenLed.Off();
        vTaskDelay(100/portTICK_PERIOD_MS);
    }
}
```

`BlueTaskB`, on the other hand, will rapidly blink the blue LED three times after successfully *taking* the semaphore, as shown below:

```
void BlueTaskB( void* argument )
{
    while(1)
    {
        // 'take' the semaphore with no timeout.
        if( xSemaphoreTake(semPtr, portMAX_DELAY) == pdPASS )
```

```
        {
            SEGGER_SYSVIEW_PrintfHost("BlueTaskB received semPtr");
            // Triple-blink the Blue LED
            for(uint_fast8_t i = 0; i < 3; i++)
            {
                BlueLed.On();
                vTaskDelay(50/portTICK_PERIOD_MS);
                BlueLed.Off();
                vTaskDelay(50/portTICK_PERIOD_MS);
            }
        }
        else
        {
            This is the code that would be executed if we timed out
        }
    }
}
```

`xSemaphoreTake`'s second parameter is the wait time. It specifies the maximum number of ticks to be in the Blocked state, while waiting for the semaphore. In our system, the wait-time `portMAX_DELAY` specifies to wait indefinitely. If a wait-time of `0` is specified, `xSemaphoreTake` returns immediately if the semaphore is not available.

The return value from `xSemaphoreTake` is `pdPASS` if the semaphore was obtained, and `pdFAIL` if the call timed out and the semaphore was not obtained. When an indefinite wait is requested, it is not necessary to process a timeout, as is shown here.



For `xSemaphoreTake()`, the wait-time is specified in ticks (SysTicks).

When the wait-time `portMAX_DELAY` is specified, its meaning depends on how `FreeRTOSConfig.h` defines `INCLUDE_vTaskSuspend`.

If `FreeRTOSConfig.h` specifies `#define INCLUDE_vTaskSuspend 1`, then `portMAX_DELAY` specifies an indefinite wait. On our system, `INCLUDE_vTaskSuspend` is `1`.

If `INCLUDE_vTaskSuspend` is `0`, then a wait-time of `portMAX_DELAY` specifies a tick count. `portMAX_DELAY` is `0xFFFFFFF` on our system, and in ticks, it would be about 50 days.

Great! Now that our code is ready, let's see what this behavior looks like.

Understanding the behaviour

Here's what this example looks like when viewed using SystemView. The top window is the **Timeline** window, and the bottom window is the **Context** window. (*Figure 8.7* is expanded in *Figures 8.7a* and *8.7b*.)

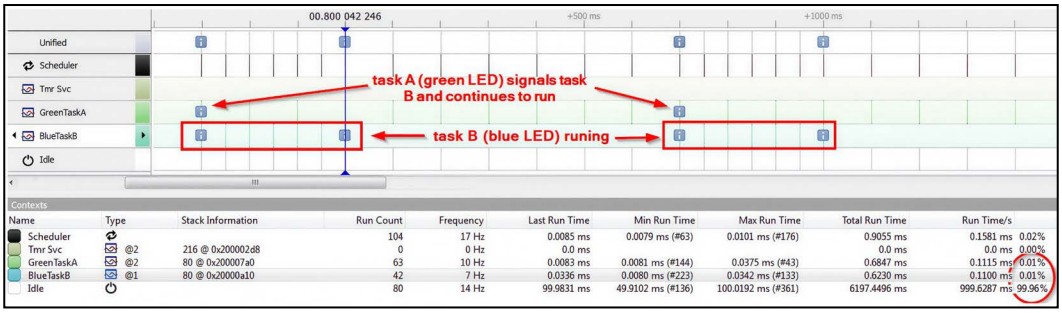


Figure 8.7: Binary semaphore, shown in the SystemView timeline

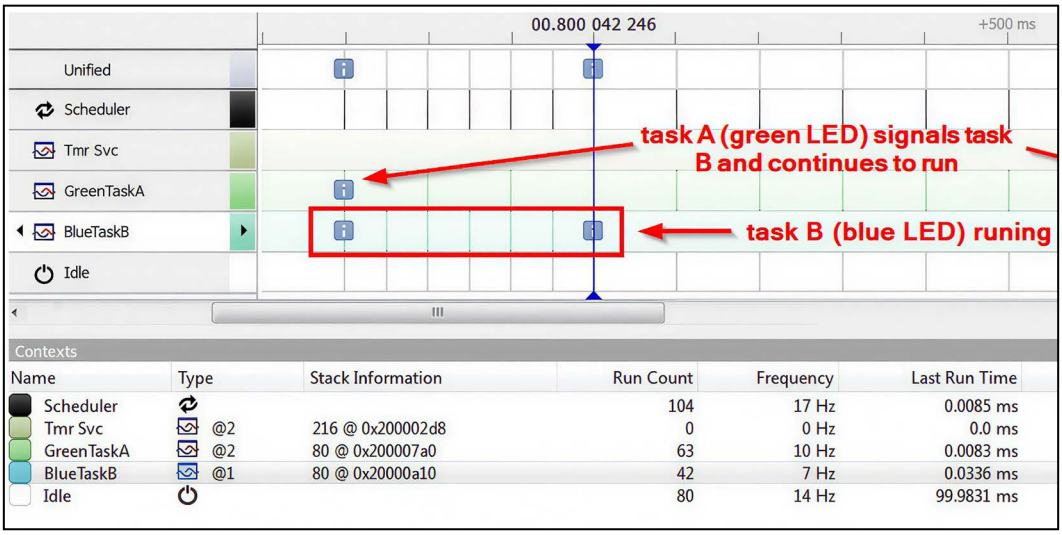


Figure 8.7a : The left side of figure 8.7

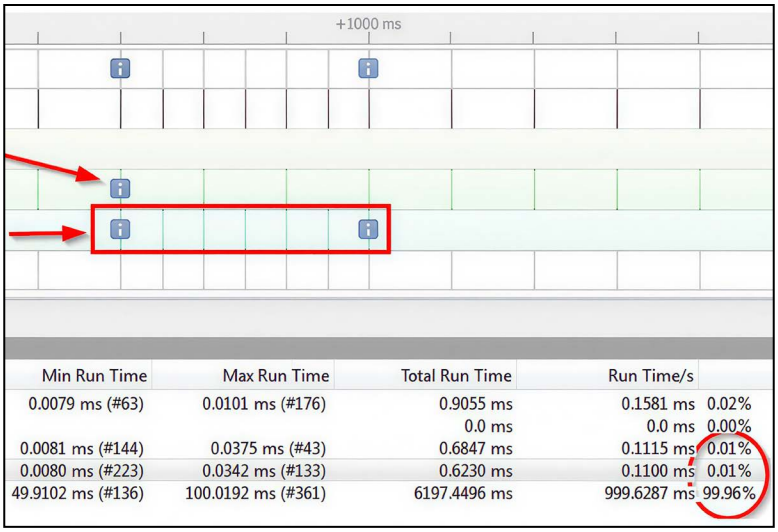


Figure 8.7b : The right side of figure 8.7

Notice the following:

- Blocking with semaphores is efficient as each task is only using 0.01% of the CPU time. This is shown in the circled column. The column is **CPU Load**, but its header isn't shown.
- When `BlueTaskB` is blocked because it is waiting on a semaphore, it won't run until the semaphore is available. This would be true even if it was the highest-priority task in the system, and no other application tasks were in the Ready state (the `Idle` task is always in the Ready or Running state).

We've observed some bugs in the Context window. For example, in the **CPU Load** column, the percentages can sum to more than 1, due to the Idle task's percentage being much too large. Those bugs appear to be in the SystemView Target Sources that are used in the example-programs. The bugs are fixed in newer versions of the Target Sources. (The Target Sources were described in chapter 5.)

Now that you've seen an efficient way of synchronizing tasks with a binary semaphore, let's have a look at another way of achieving the same behavior using polling.

Wasting cycles – synchronization by polling

Polling is a technique that is typically used in a bare-metal system that is lacking RTOS mechanisms. Let's modify our original example to replace the semaphore with a polled flag. Doing so results in the exact same behavior as when we're looking at LEDs from outside the board – the observable pattern of the LEDs is exactly the same. However, there is a big difference in how much CPU time is being used, by continuously polling the flag.

Understanding the code

The example-program is in `Src/mainPolledExample.c`. Here's the updated `GreenTaskA()` – only a single line has changed to set a flag in place of giving the semaphore.

```
volatile uint32_t flag = 0;
. . .
void GreenTaskA( void* argument )
{
    uint_fast8_t count = 0;
    while(1)
    {
        // Every 5 times through the loop, set the flag
        if(++count >= 5)
        {
            count = 0;
            SEGGER_SYSVIEW_PrintfHost("Task A (green LED) sets flag");
            flag = 1; // Set 'flag' to 1 to "signal" BlueTaskB to run
        }
        GreenLed.On();
        vTaskDelay(100/portTICK_PERIOD_MS);
        . . .
    }
}
```

Instead of calling `xSmeaphoreGive()`, we're simply setting the `flag` variable to `1`.

A similar change has been made to `BlueTaskB()`. The code to take the semaphore has been replaced with a loop that polls the flag.

```
void BlueTaskB( void* argument )
{
    while(1)
    {
        SEGGER_SYSVIEW_PrintfHost("Task B (Blue LED) starts polling on
flag");

        // Repeatedly poll on flag. As soon as it is non-zero,
        // blink the blue LED 3 times
        while(!flag);
        flag = 0;
        SEGGER_SYSVIEW_PrintfHost("Task B (Blue LED) received flag");

        // Triple blink the Blue LED
        for(uint_fast8_t i = 0; i < 3; i++);
        . . .
```

Instead of calling `xSemaphoreTake()`,we're using a `while` loop to poll on `flag`.

`flag` is a global variable, and it is set by `TaskA` and `TaskB`. `flag` must be declared as `volatile`. Otherwise, the program will not work. `volatile` informs the compiler that: a) the value in `flag` can change at any time, b) whenever `flag` is referenced in the code, the variable in memory should be used, and c) the compiler cannot perform optimizations on the variable, such as caching its value in a register. This is the customary meaning of "volatile," and it appears to be how `volatile` works with gcc in this case.

These are the only changes that are required. `BlueTaskB()` will wait to move-on (indefinitely), until `flag` is set to something other than `0`.

The tasks use the `flag` variable concurrently. Such concurrent access is possible with Cortex-M processors, because its reads and writes to `uint32_t` variables are **atomic**. And, flag is a `uint32_t` variable. So, reads and writes to `flag` cannot be interrupted by an ISR or a context-switch. (More information on atomic memory-access can be found in the ARM document, *ARMv7-M Architecture Reference Manual*, in section A3.5.3 *Atomicity in the ARM architecture*.)

Understanding the behavior

Since only a few changes were made, we might not expect there to be *that* much of a difference in terms of how the MCU is behaving, given the new code. However, the output observed with SystemView tells a different story. (*Figure 8.8* is expanded in *figures 8.8a* and *8.b*.)

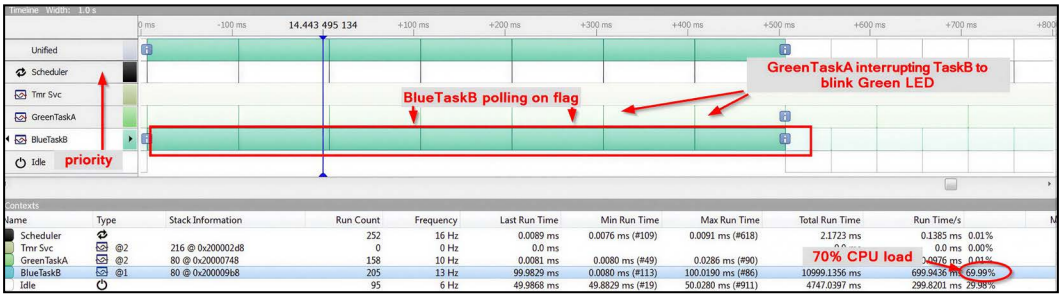


Figure 8.8 : Polling, shown in SystemView Timeline

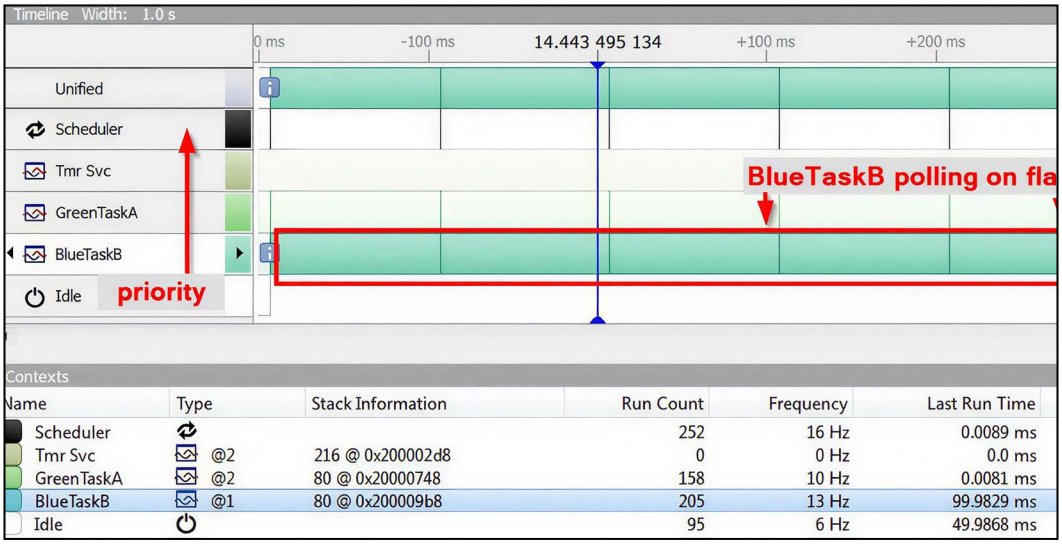


Figure 8.8a : The left side of figure 8.8

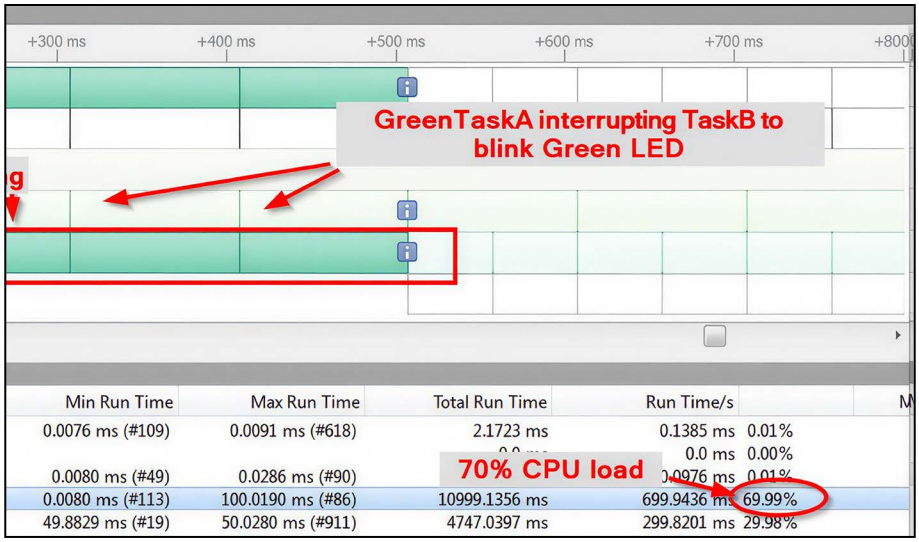


Figure 8.8b : The right side of figure 8.8

Note the following:

- `BlueTaskB` consumes processor-time while polling the value of `flag`. Consequently, `BlueTaskB` uses 70% of the system's CPU load.
- Even though `BlueTaskB` is hogging the CPU, `GreenTaskA` still runs consistently since it has a higher priority. `GreenTaskA` would be starved of CPU if it was a lower priority than `BlueTaskB`.

So, synchronizing tasks by polling on a variable *does* work as expected, but there are some side effects: increased CPU use and a strong dependency on task priorities. Of course, there are ways of reducing the CPU load of `BlueTaskB`. We could have added a delay between polling, like so:

```
while(!flag)
{
    vTaskDelay(1);
}
```

This will reduce the CPU load of `BlueTaskB` to around 5%. Beware, though, that this can potentially increase `BlueTaskB`'s polling-loop by almost 1 SysTick (1 ms on our system).

Time-bound binary semaphores

Earlier, we mentioned that one of the critical aspects of RTOSes was their ability to provide a way to time-bound operations; that is, they can guarantee a system-call doesn't stop a task from executing any longer than is desirable. **An RTOS does not guarantee the successful timeliness of an operation.** It only promises that the call will be returned within a specified amount of time.

Let's have another look at the FreeRTOS API for taking a semaphore:

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                           TickType_t xTicksToWait );
```

- `xSemaphore` is a pointer to the semaphore.
- `xTicksToWait` specifies the maximum amount of time to wait for the semaphore. If `0` is specified, `xSemaphoreTake` returns immediately if the semaphore is not available.
- The return value is `pdPASS` (the semaphore was taken in time) or `pdFALSE` (the semaphore was not taken in time).

Understanding the code

This example is in `mainSemTimeBound.c`. There are two tasks:

- `GreenTaskA`, with priority `(tskIDLE_PRIORITY + 2)`
- `TaskB`, with priority `(tskIDLE_PRIORITY + 1)`

Here, we'll be using all three LEDs on the dev-board to indicate different states:

- **Green LED**: blinked by `GreenTaskA()` at a steady 5 Hz with a 50% duty cycle.
- **Blue LED**: rapidly blinked three times by `TaskB()`, when it receives the semaphore in time.
- **Red LED**: turned on by `TaskB()` after a timeout from `xSemaphoreTake()`. Turned off by `TaskB()` when it receives the semaphore in time.



In many systems, missing a deadline can be a cause for (major) concern. It all depends on what it is you're implementing. This example is just a simple loop with a red light for when a deadline is missed. However, other systems may require that emergency procedures be taken to prevent significant failure and damage if a deadline is missed.

`GreenTaskA()` has two responsibilities:

- Blink the green LED.
- *Give* the semaphore at pseudo-random intervals.

This can be seen in the following code:

```
void GreenTaskA( void* argument )
{
    uint_fast8_t count = 0;
    while(1)
    {
        // Get a random number between 3 and 9, inclusive.
        uint8_t numLoops = StmRand(3,9);
        if(++count >= numLoops)
        {
            count = 0;
            SEGGER_SYSVIEW_PrintfHost("Task A (green LED) gives semPtr");
            xSemaphoreGive(semPtr);
        }
        greenBlink();
    }
}
```

`TaskB()` also has two responsibilities:

- Blink the blue LED (as long as the semaphore shows up within 500 ms).
- Turn on the red LED (if the semaphore doesn't show up within 500 ms). The red LED will stay on until the semaphore has successfully been taken within 500 ms of starting to wait for it.

This can be seen in the following code:

```
void TaskB( void* argument )
{
    while(1)
    {
        // 'take' the semaphore with a 500mS timeout
        SEGGER_SYSVIEW_PrintfHost("attempt to take semPtr");
        if( xSemaphoreTake(semPtr, 500/portTICK_PERIOD_MS) == pdPASS )
```

```
        {
            // Received semPtr in time
            RedLed.Off();
            SEGGER_SYSVIEW_PrintfHost("received semPtr");
            blueTripleBlink();
        }
        else
        {
            // This code is run when the semaphore wasn't taken in time
            SEGGER_SYSVIEW_PrintfHost("FAILED to receive semPtr. . .");
            RedLed.On();
        }
    }
}
```

This setup guarantees that `TaskB()` will be taking some action *at least* every 500 ms.

Understanding the behavior

When running the firmware in the **semaphoreTimeBound** build-configuration, you'll see something similar to the following figure, when using SystemView. (In SystemView's **Events List** window (top-left), ISR events are not shown here. To implement this, right-click within the Events List window, and de-select **Show ISRs**.)

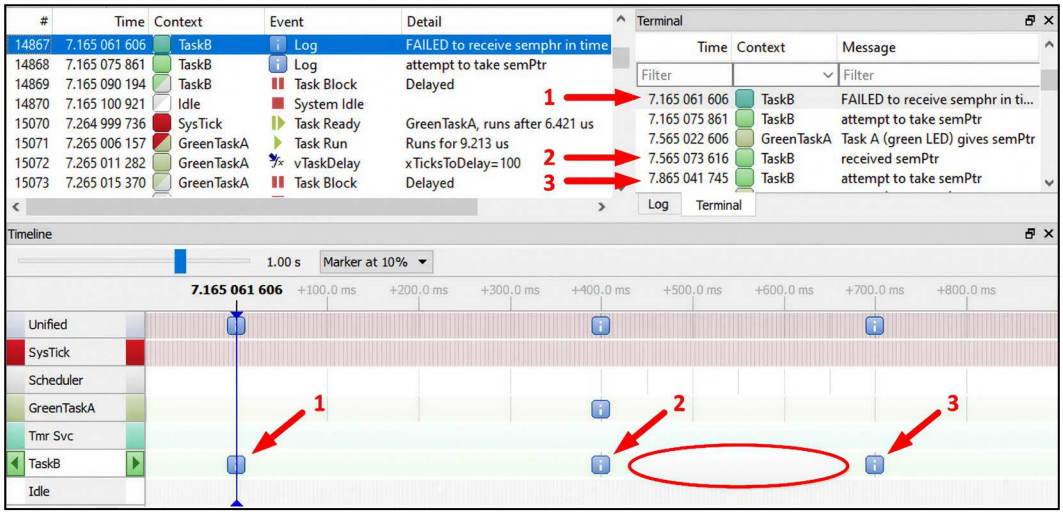


Figure 8.9 : Time-bound binary semaphore

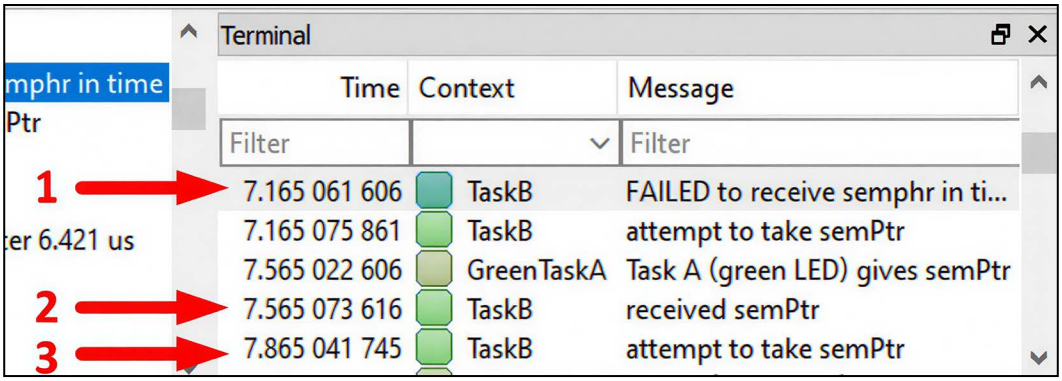


Figure 8.9a : The right side of Figure 8.9

Note the following:

1. **Marker 1** indicates `TaskB` didn't receive the semaphore within 500 ms. Notice `TaskB` promptly went back to taking the semaphore again.
2. **Marker 2** indicates `TaskB` received the semaphore within 500 ms. Looking at the graph, we can see it was actually around 400 ms. The circled area in the image is where `TaskB` blinks the blue LED.
3. **Marker 3** indicates `TaskB` goes back to waiting for the semaphore.



Log messages are indicated by blue "i" icons in the Timeline and Events List.

You'll notice that the blue LED doesn't always blink – occasionally, the red LED blinks instead. Each time the red LED blinks, this indicates that `semPtr` was not taken within 500 ms.

As an exercise, see if you can capture a red blink and track where the timeout occurred using the **Terminal** output (on the right) and the **Timeline** output (on the bottom) – how much time elapsed between when `TaskB` attempted to *take the semaphore* and when the red LED blinked? Now, modify the 500 ms timeout in the source code, compile and upload it with Ozone, and watch for the change in SystemView.

Counting semaphores

Up until this point we have been using binary semaphores, which means that the semaphore can only have a value of either 0 or 1. While binary semaphores can only have one of two values, counting semaphores can have a wider range of values. Some use-cases for counting semaphores include simultaneous connections in a communication stack or static buffers from a memory pool.

For example, let's say we have a TCP/IP stack that supports multiple simultaneous TCP sessions, but the MCU only has enough RAM to support three simultaneous TCP sessions. This would be a perfect use-case for a counting semaphore.

The counting semaphore for this application needs to be defined so that it has a maximum count of 3 and an initial value of 3 (three TCP sessions are available):

```
SemaphoreHandle_t semPtr = NULL;
semPtr = xSemaphoreCreateCounting( /*max count*/ 3, /*init count*/ 3);
if(semPtr != NULL)
```

The code that requests to open a TCP session would *take* `semPtr`, reducing its count by 1:

```
if(xSemaphoreTake( semPtr, /*timeout ticks*/ 100) == pdPASS)
{
    // Resources for TCP session are available
}
else
{
    // Timed out waiting for session to become available
}
```

Whenever a TCP session is closed, the code closing the session *gives* `semPtr`, increasing its count by 1:

```
xSemaphoreGive( semPtr );
```

By using a counting semaphore, you can control access to a limited number of available TCP sessions. By doing this, we're accomplishing two things:

- Limiting the number of simultaneous TCP sessions, thus keeping resource usage in check.
- Providing time-bound access for creating a TCP session. This means the code is able to specify how long it will wait for a session to become available.

In summary, counting semaphores are useful for controlling access to a shared-resource when more than one instance is available. Next, we'll look at a common problem that can arise in using semaphores.

Priority inversion (how not to use semaphores)

Semaphores can be used to synchronize multiple tasks. But, a limitation of semaphores is that they have no concept of task priority. This can be a problem when a higher-priority task waits on a semaphore that is being held by a lower-priority task. The higher-priority task can be inadvertently delayed by the relatively slow progress of the lower-priority task. This problem is called **priority inversion.**

This section explains priority inversion, and demonstrates it in an example-program. A solution to the problem is an enhanced form of semaphores called *mutexes*, and they are presented in a later section.

Understanding priority inversion

Let's look at a common problem that occurs when attempting to use a binary semaphore to provide mutual-exclusion functionality.

Consider three tasks, A, B, and C, where A has the highest priority, B has the middle priority, and C has the lowest priority. Tasks A and C rely on a semaphore to give access to a resource that is shared between them. Since `TaskA` is the highest-priority task in the system, it should always be running before other tasks. However, since `TaskA` and `TaskC` both rely on a resource shared between them (guarded by the binary semaphore), there is an unexpected dependency here:

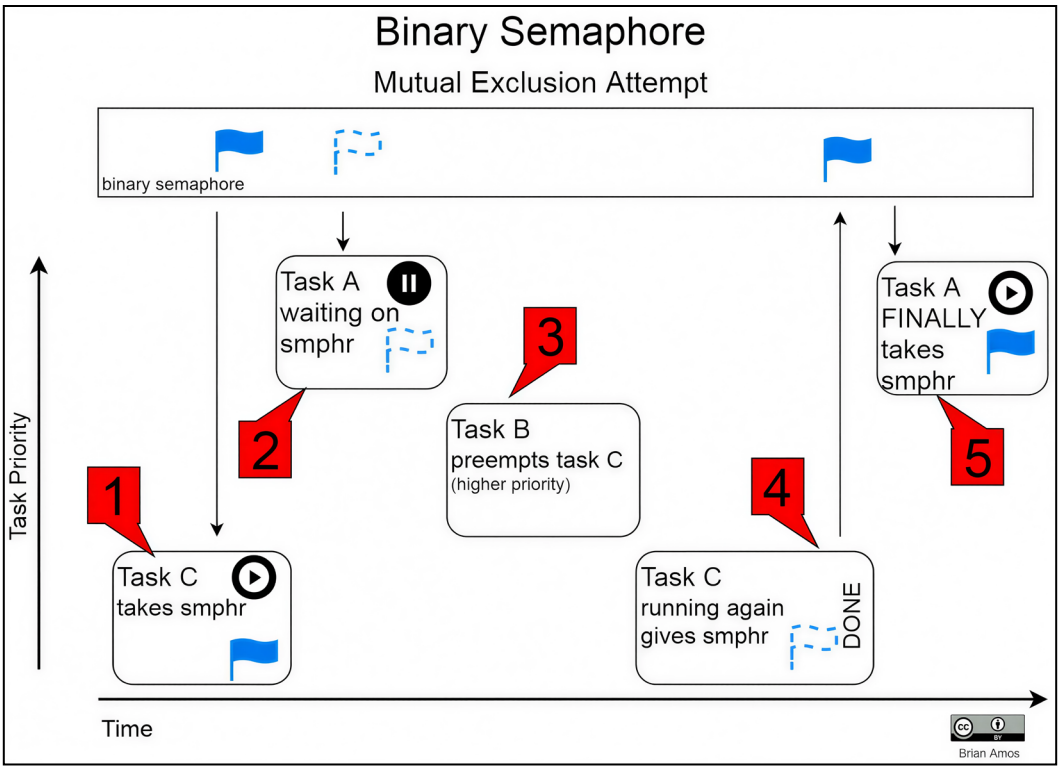


Figure 8.10 : Binary semaphore, mutual-exclusion attempt

Let's walk step-by-step through the example to see how this scenario plays out:

1. `TaskC` (the lowest-priority task in the system) acquires a binary semaphore and starts to do some work.
2. Before `TaskC` completes its work, `TaskA` (the highest-priority task) interrupts and attempts to acquire the same semaphore, but is forced to wait because `TaskC` has already acquired the semaphore.
3. `TaskB` pre-empts `TaskC` as well, because `TaskB` has a higher priority than `TaskC`. `TaskB` performs whatever work it has and then blocks.
4. `TaskC` runs through the remainder of its work with the shared-resource, at which point it gives the semaphore back.
5. `TaskA` is *finally* able to run.

`TaskA` was able to run, eventually, but not until TWO lower-priority tasks had run. `TaskC` finishing its work with the shared-resource was unavoidable (unless a design change could have been made to prevent it from accessing the same shared-resource as `TaskA`). However, `TaskB` also had an opportunity to run to completion, even though `TaskA` was waiting around and had a higher priority! This is **priority inversion** – a higher-priority task in the system is waiting to run, but it is forced to wait while a lower-priority task is running instead – the priorities of the two tasks are effectively inverted in this case.

Let's look at an example of priority inversion *in the wild* with some real code and observe the behavior first-hand!

An example-program, with priority inversion

The example-program is at `Src\mainSemPriorityInversion.c`. It is an implementation of the conceptual example we just covered. The **shared-resource** is a function that is used to blink the LEDs.



The **shared LEDs** are only an example. In practice, you'll find that when data is shared between tasks, the data often needs to be protected. There is also the chance that multiple tasks will attempt to use the same hardware peripheral, and access to that peripheral will need to be protected.

To provide some visual feedback, we'll also assign some LEDs to the various tasks. Let's have a look at the code.

Understanding the code

We'll now look at each of the three tasks and how they work.

TaskA (highest priority)

`TaskA` is responsible for blinking the green LED, but only after `semPtr` has been taken (within 200 ms of requesting it). Among the three tasks, `TaskA` has the highest priority (`tskIDLE_PRIORITY + 3`).

```
void TaskA( void* argument )
{
    . . .
    while(1)
    {
        // 'take' the semaphore with a 200mS timeout
```

```
        SEGGER_SYSVIEW_PrintfHost("attempt to take semPtr");
        if(xSemaphoreTake(semPtr, 200/portTICK_PERIOD_MS) == pdPASS)
        {
            RedLed.Off();
            receivedCounter++;
            SEGGER_SYSVIEW_PrintfHost("received semPtr: %u ",
                                      receivedCounter);
            blinkTwice(&GreenLed);
            xSemaphoreGive(semPtr);
        }
        else
        {
            // This code is called when the semaphore wasn't taken in time
            timedoutCounter++;
            SEGGER_SYSVIEW_PrintfHost(
                "FAILED to receive semPtr in time: %u ", timedoutCounter);
            RedLed.On();
        }
        // Wait for a bit to let other tasks run
        vTaskDelay(StmRand(5,30));
    }
}
```

`blinkTwice` blinks the specified LED twice. When the semaphore is taken, the amount of time it is held is the duration of `blinkTwice`, plus a negligible amount of time to run a few other instructions. If `blinkTwice` is not interrupted, it runs for a little over 100 ms.

```
static void blinkTwice( LED* led )
{
    for(uint32_t i = 0; i < 2; i++)
    {
        led->On();
        vTaskDelay(25/portTICK_PERIOD_MS);
        led->Off();
        vTaskDelay(25/portTICK_PERIOD_MS);
    }
}
```

So, when `TaskA` *takes* the semaphore, it will run for a little over 100 ms, then give the semaphore.

`TaskA` is a focal point of this example, so make sure that you have a solid understanding of the conditional statements around the semaphore being taken within the specified period. The semaphore won't always be taken in time.

TaskB (medium priority)

`TaskB` alternately delays and uses the processor. The delay is 10 to 25 ms. The spin-loop runs for a little over 30 to 75 ms of processor time. `TaskB` is the medium-priority task (`tskIDLE_PRIORITY + 2`).

```
void TaskB( void* argument )
{
    . . .
    while(1)
    {
        counter++;
        SEGGER_SYSVIEW_PrintfHost("starting iteration %u ", counter);
        vTaskDelay(StmRand(10,25));

        // Each for-loop iteration takes 1 ms of processing time.
        // The for-loop, as a whole, will use 30-75 ms of processing time.
        spinTime = StmRand(30,75);
        for (i=0; i<spinTime; i++){
            lookBusy(iterationsPerMilliSecond);
        }
    }
}
```

TaskC (lowest priority)

`TaskC` is responsible for blinking the blue LED, but only *after* the `semPtr` has been taken (within 200 ms of requesting it). `TaskC` works the same as `TaskA`, except `TaskC` blinks the blue LED, and `TaskC` does not call `vTaskDelay` directly. `TaskC` is the lowest-priority task (`tskIDLE_PRIORITY + 1`).

```
void TaskC( void* argument )
{
    . . .
    while(1)
    {
        // 'take' the semaphore with a 200mS timeout
        SEGGER_SYSVIEW_PrintfHost("attempt to take semPtr");
        if(xSemaphoreTake(semPtr, 200/portTICK_PERIOD_MS) == pdPASS)
        {
            RedLed.Off();
            receivedCounter++;
            SEGGER_SYSVIEW_PrintfHost("received semPtr: %u ",
                                      receivedCounter);
            blinkTwice(&BlueLed);
            xSemaphoreGive(semPtr);
        }
        else
        {
            // This code is called when the semaphore wasn't taken in time
            timedoutCounter++;
            SEGGER_SYSVIEW_PrintfHost(
                "FAILED to receive semPtr in time: %u ", timedoutCounter);
            RedLed.On();
        }
    }
}
```

Understanding the behavior

In our example-program, the existence of priority inversion can be seen by running the code. From the **Terminal** window, shown below, we see `TaskA` fails to *take* the semaphore about 2.0 times per second, on average, and `TaskC` about 1.2 times per second. The priority inversion can be further confirmed by making `TaskC` medium-priority, and `TaskB` low-priority. In that case, `TaskA` and `TaskC` never fail to take the semaphore.

Terminal

Time	Context	Message
Filter		Filter
27:17.535 531 551	TaskB	starting iteration 23684
27:17.595 041 852	TaskA	FAILED to receive semPtr in time: 3248
27:17.604 426 898	TaskB	starting iteration 23685
27:17.624 018 458	TaskA	attempt to take semPtr
27:17.683 531 690	TaskB	starting iteration 23686
27:17.683 590 449	TaskA	received semPtr: 6716
27:17.683 622 704	TaskC	attempt to take semPtr
27:17.751 444 486	TaskB	starting iteration 23687
27:17.810 018 403	TaskA	attempt to take semPtr
27:17.810 042 204	TaskA	received semPtr: 6717
27:17.820 516 019	TaskB	starting iteration 23688
27:17.869 326 333	TaskB	starting iteration 23689
27:17.883 041 847	TaskC	FAILED to receive semPtr in time: 1885
27:17.883 057 389	TaskC	attempt to take semPtr
27:17.920 290 736	TaskB	starting iteration 23690
27:17.920 331 968	TaskC	received semPtr: 3320

Figure 8.11 : Messages showing priority-inversion

From static code-analysis, we can see that much of what causes `TaskA` to not take the semaphore on time is `TaskB`'s spin loop, and `TaskB` having a higher-priority than `TaskC`.

For `TaskA` to fail to take the semaphore in time, `TaskC` has to hold the semaphore for 200 ms, or more. If `TaskC` were to take the semaphore and not be interrupted, it would hold the semaphore for a little over 100 ms. So, what could cause `TaskC` to hold the semaphore for more than 100 ms?

A simple case to consider is when `TaskC` takes the semaphore, and just after that, `TaskA` attempts to take the semaphore. `TaskA` will then block. Next, what causes `TaskC` to hold the semaphore for more than 100 ms is `TaskB` running when `TaskC` is in the Ready state. This primarily occurs when `TaskB` is running its spin-loop, and `TaskC` is not blocked due to `vTaskDelay`. From `TaskB`'s code, we see that when `TaskA` is blocked, `TaskB` will spin a lot: at least 3/4 of the time, on average.

Observant readers may have noticed that `TaskA` and `TaskB` both set the red LED, but there's no semaphore to ensure only one task can do that at a time. Such a semaphore would be needed if: a) a task could start the operation of setting the red LED, but not complete it due to a context switch, and b) a subsequent task could attempt to set the red LED, but fail due to the prior operation not completing. From the program's logic, it seems unlikely that two tasks would ever set the red LED concurrently like this. However, it's difficult to prove that could never happen.

Fortunately, the code that changes the LED's state is an atomic operation. So, once that operation starts, a context switch cannot occur before the operation completes. Therefore, a semaphore is not needed to ensure only one task can set the red LED at a time. (Internally, the LED is set by the `HAL API HAL_GPIO_WritePin`. According to the *HAL User Guide*, that API is an atomic operation, in that an ISR cannot occur while the LED is being set. Presumably, this would also prevent a context switch from occurring then, since a context switch is started via an ISR.)

Next, we'll look at mutexes – an alternative to binary semaphores, which are not as prone to priority inversion.

Mutexes

The term **mutex** is shorthand for **mutual exclusion**. In the context of shared-resources and tasks, mutual exclusion means that if a task is using a shared-resource, then that task is the only task that is permitted to use the resource – all others will need to wait. If this sounds a lot like a binary semaphore, that's because it is.

This section explains what mutexes are, and how to work with them. Also, an example-program is presented.

Understanding mutexes

In FreeRTOS, mutexes are binary semaphores with one important additional feature. That feature is **priority inheritance** – mutexes have the ability to temporarily change the priority of a task to avoid causing major delays in the system. This plays out when the scheduler finds that a high-priority task is attempting to acquire a mutex already held by a lower-priority task. In this specific case, the scheduler will temporarily increase the priority of the lower task until it releases the mutex. At that point, the priority of the lower task will be set back to what it was prior to the **priority inheritance**. Let's take a look at an example. It is just like the example presented for priority inversion, but it uses a mutex instead of a binary semaphore.

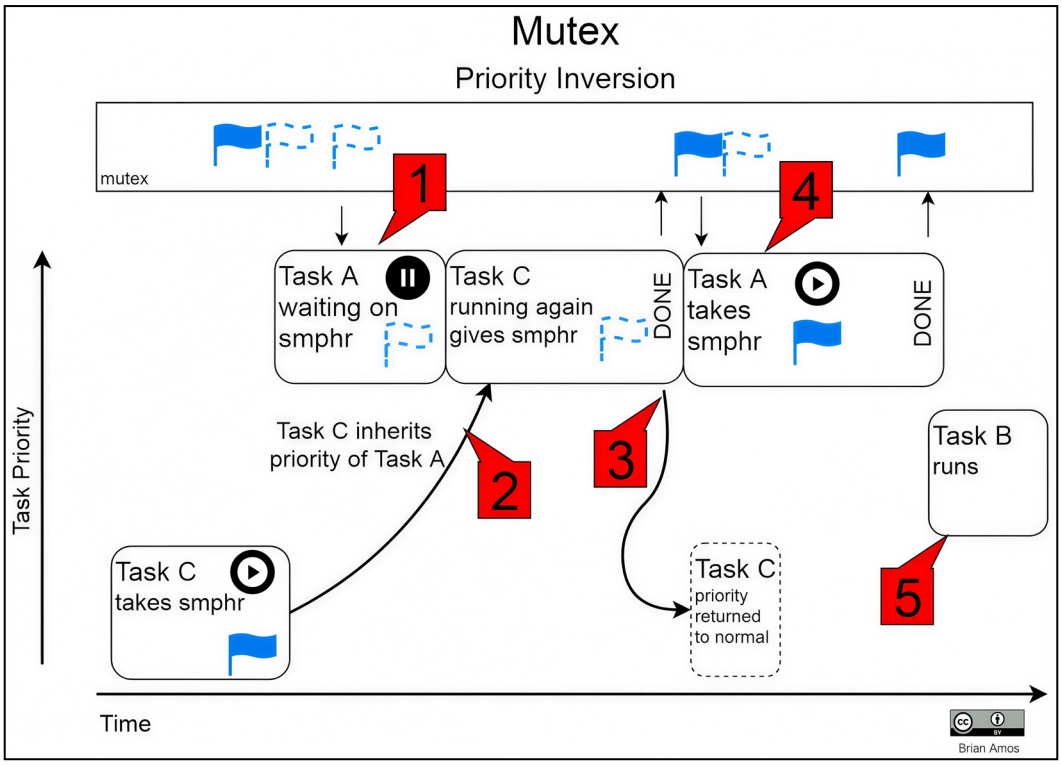


Figure 8.12 : How a mutex avoids priority-inversion

Let's walk step-by-step through the example to see how this scenario plays out:

1. `TaskA` is waiting for `TaskC` to return the mutex.
2. The priority of `TaskC` is brought up to be the same as that of the higher-priority `Task A`. `TaskC` runs to completion since it holds the mutex and is a high priority task.

3. `TaskC` returns the mutex. `TaskC`'s priority is demoted to what it was before `TaskC` was holding the mutex.
4. `TaskA` takes the mutex and completes its work.
5. `TaskB` is allowed to run.

Elevating `TaskC`'s priority like this can be a major concern or no big deal, depending on how long `TaskC` is using the shared-resource and how time-sensitive `TaskA` is. Timing analysis can be performed to ensure that `TaskA` is still meeting deadlines. However, analyzing all possible causes of priority inversion and all other high-priority asynchronous events could prove to be challenging. At a minimum, the user should make use of the built-in timeouts provided for taking mutexes, and perform a suitable alternative action if a mutex times-out.

An example-program, fixing priority-inversion

Let's have another try at protecting the shared-resource `blinkTwice()`, but this time we'll use a mutex instead of a semaphore. Using a mutex should help *minimize* priority-inversion. The mutex will prevent the mid-priority task from running when the lowest-priority task has the mutex and is able to run.

Understanding the code

The example-program is `mainMutexExample.c`. It is almost identical to the prior example, but for two significant differences:

- We'll use `xSemaphoreCreateMutex` instead of `xSemaphoreCreateBinarySemaphore`.
- No initial `xSemaphoreGive()` call is required since the mutex will be initialized with a value of 1. Mutexes are designed to be taken only when needed and then given back.

Here's our new code that creates the mutex:

```
mutexPtr = xSemaphoreCreateMutex();
assert_param(mutexPtr != NULL);
```

There are some additional changes related to the variable-name change from `semPtr` to `mutexPtr`, but otherwise, the code is the same.

Understanding the behavior

Using Ozone, load `Chapter8_mutexExample.elf` and run the MCU. Here's what to expect when looking at the board:

- You'll see double-blinking green and blue LEDs. The LED-blinks of each color will not overlap one another, thanks to the mutex.
- Red LED blips will only occur every once in a while. This reduction is caused by `TaskB` not being allowed to take priority over `TaskC`, when `TaskC` has the mutex and is able to run. This is a lot better than before, but why are we still seeing the red LED occasionally?

Tests were run for the two example-programs, to contrast priority-inversion (with a semaphore) and the use of a mutex. The results are in the table below, and show the rates at which the semaphore and mutex were taken, per second. The example-programs were run for 20 minutes. The tests reveal:

- `TaskA` never timed out when attempting to take the mutex. Using a binary semaphore, `TaskA` timed out about 30% of the time when attempting to take the semaphore.
- The rate `TaskA` successfully takes the mutex is about 50% more than the rate at which `TaskA` takes the semaphore with priority-inversion.
- The rate `TaskC` successfully takes the mutex is about 60% more than the rate at which `TaskC` takes the semaphore with priority-inversion.
- The rate `TaskC` timed out in taking the mutex is about the same as the rate at which `TaskC` timed out in taking the semaphore with priority-inversion.

Example-program	Outcome	TaskA	TaskC
Priority inversion	Taken	4.1/sec	2.0/sec
	Timed out	2.0/sec	1.2/sec
Mutex	Taken	6.2/sec	3.3/sec
	Timed out	0/sec	1.2/sec

Table 8.1 : Contrasting priority-inversion (with a semaphore) and the use of a mutex

It makes sense that `TaskA` never times-out when attempting to take the mutex, because if `TaskA` is waiting on the mutex, `TaskC` is elevated to the highest priority. And, in that case, `TaskC` will hold the mutex for a little over 100 ms, at most.

Using the mutex substantially improves all of the performance measures in the table, except for the `TaskC` timeout, which stayed the same. The reason is that for `TaskC` to time out, in both example-programs, `TaskC` first has to request the semaphore or mutex while `TaskA` is holding it. While `TaskC` is blocked and waiting for the semaphore or mutex, both example-programs work exactly the same. The tasks and their priorities are identical. So, the likelihood of `TaskC` timing-out is the same.

Testing showed that `TaskA` gets the mutex almost twice as often as `TaskC`. So, there must be scenarios where `TaskA` takes the mutex twice in a row. For this to happen, how would `TaskA` take the mutex, then give the mutex, but `TaskC` not get the mutex? One reason is that if multiple tasks have requested the mutex, it is given to the highest-priority task among them that is able to run. (See *Mastering the FreeRTOS Real Time Kernel*, section *Mutexes and Task Scheduling*).

The following scenario illustrates that process:

1. `TaskA` takes the mutex and calls `blinkTwice`, which calls `vTaskDelay`.
2. `TaskA` enters the Blocked state.
3. `TaskC` attempts to take the mutex, and enters the Blocked state to wait for the mutex.
4. `TaskA` and `TaskB` run, as well as the Idle task.
5. `TaskA` gives the mutex.
6. `TaskA` calls `vTaskDelay` and enters the Blocked state.
7. `TaskB` resumes its spin-loop.
8. `TaskA`'s delay ends.
9. `TaskA` resumes, attempts to take the mutex, and is given the mutex (`TaskC` had requested the mutex before `TaskA`, but `TaskA` has higher priority).
10. `TaskC` remains in the Blocked state, waiting for the mutex.

Overall, by using a mutex, we've improved our condition. `TaskA`, which is the highest-priority task, isn't missing any mutexes. Also, the rate `TaskA` and `TaskB` obtain the mutex is greater than the rate `TaskA` and `TaskB` obtain the semaphore with priority inversion. But what are some best practices to follow when using mutexes? Read on to find out.

Avoiding mutex-acquisition timeouts

While mutexes *help* to provide protection against some priority inversion, we can take an additional step to make sure the mutex isn't used longer than needed. The section of code that's protected by the mutex is referred to as a **critical section**:

```
if(xSemaphoreTake(mutexPtr, 200/portTICK_PERIOD_MS) == pdPASS)
{
    // Critical section is here
    // KEEP THIS AS SHORT AS POSSIBLE
    xSemaphoreGive(mutexPtr);
}
```

Taking steps to ensure this critical section is as short as possible will help in a few areas:

- Less time in the critical section makes the shared data more available. The less time a mutex is being held, the more likely it is that another task will gain access in time.
- Minimizing the amount of time low-priority tasks hold mutexes also minimizes the amount of time they spend in an elevated priority.
- If a low-priority task is blocking a higher-priority task from running, the high-priority task will have less variability (also known as jitter) in how quickly it is able to react to events.

Avoid the temptation to acquire a mutex at the beginning of a long sequence of code, and return the mutex at the end. Instead, just hold the mutex when it is needed. For example:

```
if(xSemaphoreTake(mutexPtr, 200/portTICK_PERIOD_MS) == pdPASS)
{
    //critical section starts here
    uint32_t aVariable, returnValue;
    aVariable = PerformSomeOperation(someOtherVarNotProtectedbyMutexPtr);
    returnValue = callAnotherFunction(aVariable);

    protectedData = returnValue;
    //critical section ends here
    xSemaphoreGive(mutexPtr);
}
```

The preceding code can be rewritten to minimize the critical section. This still accomplishes the same goals as providing mutual exclusion for `protectedData`, but the amount of time the mutex is held is reduced:

```
uint32_t aVariable, returnValue;
aVariable = PerformSomeOperation(someOtherVarNotProtectedbyMutexPtr);
returnValue = callAnotherFunction(aVariable);

if(xSemaphoreTake(mutexPtr, 200/portTICK_PERIOD_MS) == pdPASS)
{
    //critical section starts here
    protectedData = returnValue;
    //critical section ends here
    xSemaphoreGive(mutexPtr);
}
```



In the preceding examples, there were no `else` statements listed in case the action didn't complete in time. Remember, it is extremely important that the consequences of a missed deadline are understood and that the appropriate action is taken. If you *don't* have a good understanding of the required timing (and the consequences of missing it), then it is time to get the team together for a discussion.

The longer a mutex is held, the more it slows down the tasks that wait for the mutex. However, it can be challenging to figure out when to hold a mutex so that it is held for the least amount of time. The tasks' concurrent interactions can be extremely complex, making it difficult to determine this minimal mutex-use, and to ensure it does not have bugs. There is often a trade-off between a) holding mutexes for the least amount of time possible, to improve performance, and b) holding mutexes longer than might be necessary, to keep the system simple and to have greater certainty the mutex-use is correct.

A similar trade-off was encountered in the earlier section *Priority inversion…*. There, two tasks concurrently access a red LED. However, the LED-access was not protected by a semaphore. It was explained that the operations on the LED appeared to be atomic, so a semaphore was not needed. However, determining those operations are atomic was based on a somewhat speculative understanding of FreeRTOS's internals.

To be safe, a semaphore could have been used to protect the LED-access. There would be greater certainty the code is correct, as that solution doesn't rely on a speculative understanding of complex system-internals. On the other hand, using a semaphore to protect the LED would make the system slower (especially if tasks had to wait for the semaphore).

Now that we have a basic understanding of mutexes, we'll take a look at how they can be used to protect data that's being shared across multiple tasks.

Race conditions

Race conditions can occur when multiple tasks use a shared-resource, such as memory. A race condition refers to timing errors or sequencing errors in the tasks' access of the shared-resource. For example, with memory access, race conditions can result in invalid data being read or stored.

In this section, we'll just consider resources that are shared by tasks. Resources can also be shared between ISRs, and between tasks and ISRs. The principles presented here can be extended to those other scenarios.

Using mutexes and semaphores to avoid race conditions

When do we need to use mutexes and semaphores? When multiple tasks share a resource, a mutex or a semaphore is often needed. They are used to limit resource-access to one task at a time, or to a maximum number of tasks at one time. A typical application is when tasks share a set of variables, as with a `struct`. A semaphore or mutex can be used to ensure that a) only one task can be in the process of changing the data, and b) tasks do not read the data when it is in the process of being changed, and vice versa.

Standard binary semaphores can be used for resource protection, and in some special cases semaphores can be preferable (such as semaphores being accessed from ISRs). However, you must understand how waiting on the semaphore will affect the system.



We'll see an example of a semaphore being used to protect a shared-resource in *Chapter 10, Drivers and ISRs.*

A race-condition example

In our previous mutex example, the shared-resource being protected by the mutex was the function `blinkTwice`. It double-blinks an LED specified by the caller. Only one task can call `blinkTwice` at a time, and it performs the entire double-blink before it can be called again.

Let's take a look at why mutual exclusion is important using a more realistic example: accessing data stored for a multi-axis accelerometer. The data is stored in a `struct`, as shown below. It stores readings for the X, Y, and Z axes. In a high-speed environment, it is important for all three readings to be correlated with one another to accurately determine the device's movement over time.

```
typedef struct
{
    uint16_t X;
    uint16_t Y;
    uint16_t Z;
} AccelReadings;
AccelReadings sharedData;
```

In the real world, you'll often find shared data-structures and hardware peripherals are among the most common resources that need to be protected from race conditions.

When a structure contains multiple pieces of data that must be correlated with one another, operations on the `struct` typically must be **atomic**. An operation on a collection of data is **atomic** if the operation is guaranteed to complete before another task accesses the data. That is, while the operation is in progress, a context switch cannot occur.

We'll first look at two tasks that access the struct `SharedData`, but the operations are not atomic. The resultant problems will be explained, then a solution will be presented using a mutex. (The code is shown here, but it's not among the chapter's implemented example-programs.)

`Task1()` is responsible for updating the data in the structure:

```
void Task1( void* args)
{
    uint16_t newXValue, newYValue, newZValue;
    while(1)
    {
        getNewValues(&newXValue, &newYValue, &newZValue);
        sharedData.X = newXValue;
        sharedData.Y = newYValue;
        sharedData.Z = newZValue;
    }
}
```

On the other hand, `Task2()` is responsible for reading the data from the structure:

```
void Task2( void* args)
{
    uint16_t myX, myY, myZ;
    while(1)
    {
        myX = sharedData.X;
        myY = sharedData.Y;
        myZ = sharedData.Z;
        calculation(myX, myY, myZ);
    }
}
```

If one of the readings isn't properly correlated with the others, we'll wind up with an incorrect estimation of the device's movement. `Task1` may be attempting to update all three readings, but in the middle of it, `Task2` comes along and attempts to read the values. As a result, `Task2` receives an incorrect representation of the data because it was in the middle of being updated:

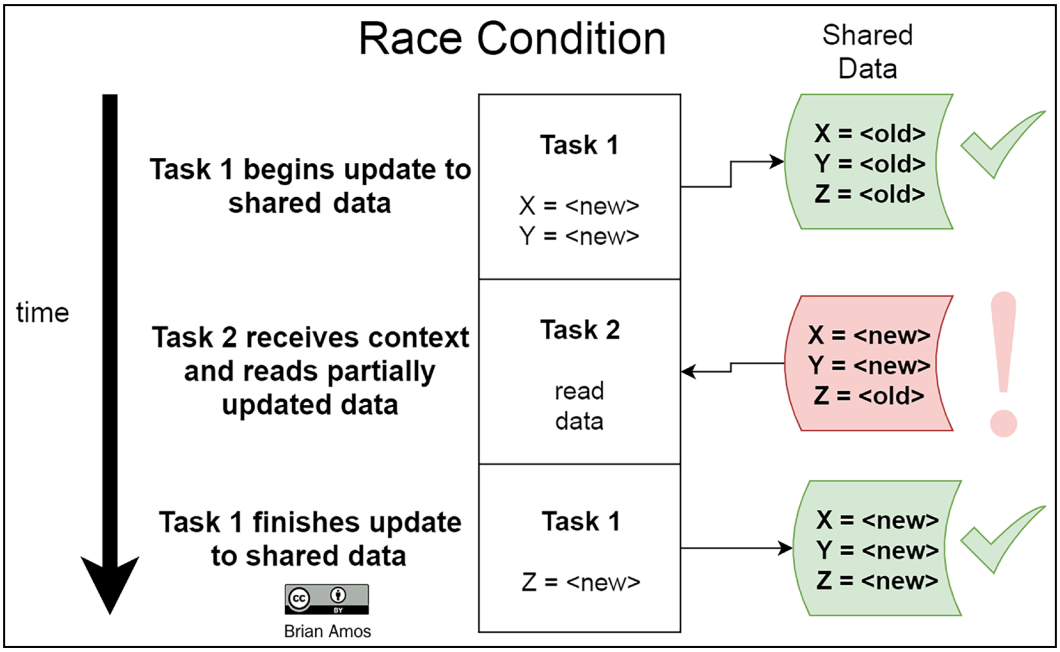


Figure 8.13 : Race condition

The solution, shown below, is for each task to use a mutex, when accessing the shared data. In each task, its code that accesses the shared data is identified as a **critical-section. The** mutex is used to make each task's critical-section be an atomic-operation.

```
void Task1( void* args)
{
    uint16_t newXValue, newYValue, newZValue;
    while(1)
    {
        // Keeping the critical-section short
        getNewValues(&newXValue, &newYValue, &newZValue);

        if(xSemaphoreTake(mutexPtr, timeout) == pdPASS)
        {
            // Critical-section start
            sharedData.X = newXValue;
            sharedData.Y = newYValue;
            sharedData.Z = newZValue;
            // Critical-section end
            xSemaphoreGive(mutexPtr);
        }
        else { /* report failure */}
    }
}
```

It is important to wrap the read accesses as well:

```
void Task2( void* args)
{
    uint16_t myX, myY, myZ;
    while(1)
    {
        if(xSemaphoreTake(mutexPtr, timeout) == pdPASS)
        {
            // Critical-section start
            myX = sharedData.X;
            myY = sharedData.Y;
            myZ = sharedData.Z;
            // Critical-section end
```

```
            xSemaphoreGive(mutexPtr);

            // Keeping the critical-section short
            calculation(myX, myY, myZ);
        }
        else{ /* report failure */ }
    }
}
```

Atomic-operations are said to **serialize** access to a resource, such as data or a function. That is, only one task can perform the atomic-operation at a time, and the atomic-operations do not overlap. The next atomic-operation cannot start until the prior atomic-operation completes.

Atomic-operations can be categorized based on the number of instructions in the atomic-operation. **Primitive atomic-instructions** are a single instruction that executes as an atomic-operation. An example is the read and write operations on the variable `flag`, in the example-program on polling. Another example is the operations for turning the red LED on and off, in the example-program on priority inversion. In contrast, **high-level atomic-operations** create a critical-section for multiple instructions. An example was just shown, where a mutex was used to create a critical-section, to maintain **coherence** in a collection of data.

Next, we'll take another look at inter-task synchronization. Semaphores were used for this previously, but what if your application calls for actions to occur at a consistent rate? FreeRTOS software-timers are one possible solution.

Software timers

This section describes software timers, and it presents an example-program that uses software timers. Also, guidelines are presented for using software timers.

Understanding software timers

Just like the name states, **software timers** are timers that are implemented with software. In MCUs, it is extremely common to have many different hardware-peripheral timers available. These are often high resolution and have many different modes and features that are used to offload work from the CPU. However, there are two downsides to **hardware timers**:

- Since hardware timers are part of the MCU, you'll need to create an abstraction above them to prevent your code from becoming tightly coupled to the underlying MCU hardware. Different MCUs will have slightly different implementations for timers. Because of this, it is easy for code to become dependent on the underlying hardware.

- Hardware timers will generally take more development time to set-up than using the software-based timer that has already been provided by the RTOS.

Software timers alleviate this coupling by implementing multiple timer-channels via software, rather than hardware. So, instead of an application being dependent on specific hardware, the application can be used (without modification) on any platform the RTOS supports, which is extremely convenient.

You may have noticed a task called `TmrSvc` in the SystemView Timeline. This is the Software-Timer Service Task. Software timers are implemented as a FreeRTOS task, using many of the same underlying primitives that are available. They have a few configuration options, all of which can be set in `FreeRTOSConfig.h`:

```
/* Software timer definitions. */
#define configUSE_TIMERS 1
#define configTIMER_TASK_PRIORITY ( 2 )
#define configTIMER_QUEUE_LENGTH 10
#define configTIMER_TASK_STACK_DEPTH 256
```

In order to have access to software timers, `configUSE_TIMERS` must be defined as 1. As shown in the preceding snippet, the timer-task's priority, as well as its queue length (number of available timers) and stack depth, can all be configured.

But, you might be thinking: "software timers are a FreeRTOS feature – why do I need to worry about stack depth?!"

There's one thing to keep in mind with software timers: the code that's executed when the timer fires is executed inside the context of the Software-Timer Service Task. This means two things:

- Each callback function executes on the `TmrSvc` task's stack. Any RAM (that is, local variables) that's used in the callback will come from the `TmrSvc` task.
- Any long actions that are performed will block other software-timers from running. So, treat the callback-function you pass to the software-timer as being similar to an ISR – don't deliberately delay the task, and keep everything as short as possible.

The best way to get familiar with software timers is to actually use them in a real system.

An example-program, with one-shot and repeat timers

To see software-timers in action, let's have a look at the examples in `mainSoftwareTimers.c`. There are two types of software timers: **one-shot** and repeat. Each type is implemented in the example-program, as described in the following sections.

Understanding the code

This section presents the code for one-shot and repeat timers.

One-shot timers

A **one-shot** is a timer that fires only *one* time. These types of timers are common in both hardware and software and come in very handy when a fixed delay is desired. A one-shot timer can be used when you wish to execute a *short* piece of code after a fixed delay, without blocking the calling code by using `vTaskDelay()`. To set-up a one-shot timer, a timer callback-function must be created, as well as the timer. This is done in the example-program, as described below:

1. Define a callback function whose address can be passed to `xTimerCreate()`. This function is executed when the timer fires. Keep in mind that the callback is executed within the timer task, so it needs to be non-blocking!

```
void oneShotCallBack( TimerHandle_t xTimer )
{
    SEGGER_SYSVIEW_PrintfHost("oneShotCallBack:  blue LED off");
    BlueLed.Off();
}
```

2. Create the timer by calling **xTimerCreate**. The second parameter specifies the time period until the callback-function is run. The third parameter specifies whether the timer is one-shot or repeating (repeating timers *auto-reload* in FreeRTOS). `pdFalse` specifies a one-shot timer.

```
TimerHandle_t oneShotHandle =
  xTimerCreate( "myOneShotTimer",          // Name for timer
                2200/portTICK_PERIOD_MS, // Period of timer in ticks
                 pdFALSE,                  // Auto-reload flag
                NULL,                    // Unique ID for timer
                oneShotCallBack);        // Callback function
```

3. Perform some due diligence checks to make sure the timer was created successfully by checking that the returned handle is not `NULL`.

```
assert_param(oneShotHandle != NULL);
```

4. The timer is started by calling `xTimerStart()`.

```
xTimerStart(oneShotHandle, 0);
```



Remember that the code that executes inside the software timer must be kept short. All software-timer callbacks are serialized (if one callback performs long operations, it could potentially delay others in starting).

Repeat timers

Repeat timers are similar to one-shot timers, but instead of getting called only *once*, they get called *repeatedly*. Since repeat timers are executed within the `TmrSvc` task, they can provide a lightweight alternative to tasks for short, non-blocking functions that need to be run periodically.

The steps for setting up a repeat timer are largely the same as they were for one-shot timers. The second parameter specifies the time period until the first call to the timer, and between calls to the timer. For the third parameter, `pdTRUE` specifies a repeat timer:

```
TimerHandle_t repeatHandle =
    xTimerCreate( "myRepeatTimer",          // Name for timer
                  500 /portTICK_PERIOD_MS, // Period of timer in ticks
                  pdTRUE,                   // auto-reload flag
                  NULL,                     // Unique ID for timer
                  repeatCallBack);          // Callback function
assert_param(repeatHandle != NULL);
xTimerStart(repeatHandle , 0);
```

The repeating timer's callback function toggles the green LED:

```
void repeatCallBack( TimerHandle_t xTimer )
{
    static uint32_t counter = 0;
    SEGGER_SYSVIEW_PrintfHost("repeatCallBack:  toggle Green LED");
    if(counter++ % 2)
    {
        GreenLed.On();
```

```
    }
    else
    {
        GreenLed.Off();
    }
}
```



In the preceding code, a static variable is used for the `counter` variable so that its value persists across function calls, while still hiding the variable from all the code outside of the `repeatCallBack()` function.

In the example-program, the timers generate SystemView messages. Messages are generated less than 3 seconds after the timers start. So, for the messages to be displayed, the timers have to be started in a task, and not in `main()`. The reason is that the SystemView app's Record-mode has to be started after `main()` calls `vTaskStartScheduler`. The SystemView app only displays messages generated after Record-mode is started.

The timers are created and started in the task `taskStartTimers`. When the task starts, it first spins until the SystemView app is started, in Record mode. The timers are then created and started.

```
void taskStartTimers( void* argument )
{
    // Indicate taskStartTimers started
    RedLed.On();
    // Spin until the user starts the SystemView app, in Record mode
    while(SEGGER_SYSVIEW_IsStarted()==0){
        lookBusy(iterationsPerMilliSecond);
    }
    RedLed.Off();

    // Create and start timers
    . . .
```

Understanding the behavior

After starting SystemView Record mode, you'll see the blue LED turn on. The timers are then started. The Terminal window (below) shows the timers running. The SystemView timestamps are relative to when Record-mode started. We can see the green LED is toggled every 500 ms, by the repeat timer. The one-shot timer turns off the blue LED 2.2 seconds after it was started.

Terminal

Time	Context	Message
Filter		Filter
0.000 786 824	Idle	taskStartTimers: starting
0.000 805 597	Idle	taskStartTimers: blue LED on
0.000 845 639	Idle	taskStartTimers: one-shot timer started (turns off blue LED)
0.000 899 606	Idle	taskStartTimers: repeating-timer started (blinks the green LED)
0.000 933 148	Idle	taskStartTimers: deleting itself
0.500 039 454	Tmr Svc	repeatCallBack: toggle Green LED
1.000 039 361	Tmr Svc	repeatCallBack: toggle Green LED
1.500 039 338	Tmr Svc	repeatCallBack: toggle Green LED
2.000 039 463	Tmr Svc	repeatCallBack: toggle Green LED
2.200 037 023	Tmr Svc	oneShotCallBack: blue LED off
2.500 039 329	Tmr Svc	repeatCallBack: toggle Green LED
3.000 039 407	Tmr Svc	repeatCallBack: toggle Green LED

Figure 8.14 : Messages from the software timers

This same information is also available in the SystemView Timeline (below):

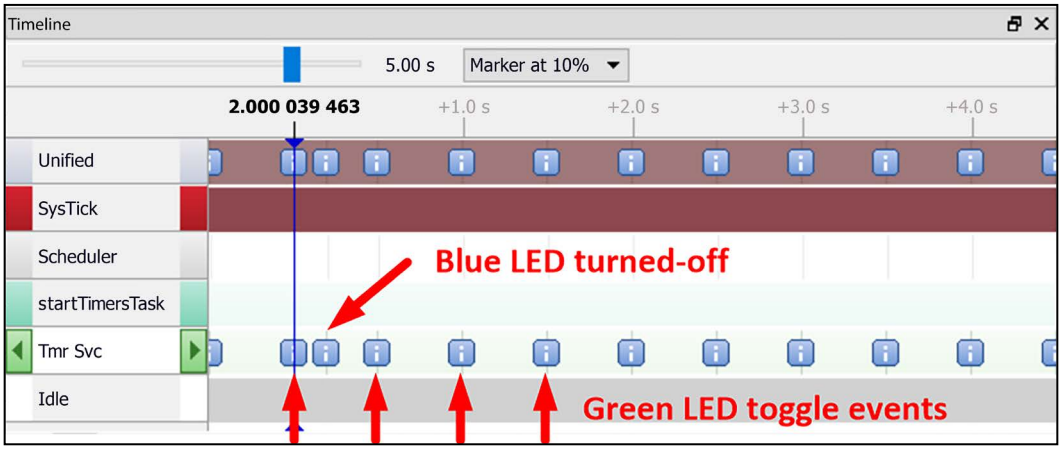


Figure 8.15: Messages from the software timers

Now that we know how to set-up software timers and understand their behavior, let's discuss when they can be used.

Software-timer guidelines

Software timers can be very useful, especially since they're so easy to set up. They are also fairly lightweight because of the way they have been coded in FreeRTOS – they don't require significant code or CPU resources.

Example use cases

These are common use-cases for software-timers:

- Periodically perform an action (auto-reload mode). For example, a timer callback-function could give a semaphore to a reporting task to provide periodic updates about the system.
- Perform an event only once at some point in the future, without blocking the calling task in the meantime (which would be required if `vTaskDelay()` was used instead).

Considerations

Keep these considerations in mind:

- The priority of the timer service-task can be configured in `FreeRTOSConfig.h` by setting `configTIMER_TASK_PRIORITY`.
- Timers can be modified after being created. They can also be restarted and deleted.
- Timers can be created statically to avoid dynamic allocation from the FreeRTOS heap (similar to static task creation).
- All callbacks are executed in the Software-Timer Service Task – they must be kept short and not block!

Limitations

So, what's not to love about software timers? Not too much, as long as the following are kept in mind:

- **Jitter**: Since the callbacks are executed within the context of a task, their exact execution time will depend on all the interrupts in the system, as well as any higher-priority tasks. FreeRTOS allows this to be tuned by adjusting the priority of the timer task being used (which must be balanced with the responsiveness of other tasks in the system).
- **Single Priority**: All software timer callbacks execute inside the same task.
- **Resolution**: A software timer's resolution is only as precise as the FreeRTOS tick rate (defined as 1 ms for most ports).

If lower jitter or higher resolution is required, it probably makes sense to use a hardware timer with ISRs, instead of software timers.

Summary

In this chapter, we covered many different aspects of synchronizing tasks and protecting shared data between tasks. We also covered semaphores, mutexes, and software timers. Then, we got our hands dirty by writing some code for each of these types and took a deep dive into analyzing the code's behavior using our Nucleo development board and SystemView.

Now, you have some tools at your disposal for solving synchronization problems, such as one task notifying another that an event has occurred (semaphores). This means you're able to safely share data between tasks by properly wrapping access in a mutex. You also know how to save a bit of RAM when performing simple operations, that is, by using software timers for small periodic operations, instead of dedicated tasks.

In the next chapter, we'll cover more crucial RTOS primitives that are used for inter-task communication and that provide the foundations for many RTOS-based applications.

Further reading

- Phillip Koopman on race conditions (lecture slides and video):`https://course.ece.cmu.edu/~ece642/`

9

Intertask Communication

Now that we're able to create tasks, it's time to start passing data between them. After all, you don't often run into systems that have a bunch of parallel tasks operating completely independently of one another; normally, you will need to pass data between the system's tasks.

In FreeRTOS, data can be sent between tasks by using **queues** and **task notifications**. First, we'll describe queues conceptually. Next, we'll show several ways queues can be used in FreeRTOS, and illustrate those uses with example-programs. We'll then move on to task notifications and learn why we should use them, and when.

The chapter's example-programs are in the IDE-project `Chapter_09`. There are four example-programs, and each has its own build configuration.

We will be covering the following topics:

- Understanding RTOS queues
- Passing data through queues by value
- Passing data through queues by reference
- Task notifications

Understanding RTOS queues

Queues are quite simple in concept, but they are also extremely powerful and flexible, especially if you've traditionally programmed on bare metal with C. At its heart, a queue is simply a circular buffer. However, this buffer contains some very special properties, such as native multi-thread safety, the flexibility for each queue to hold any type of data, and waking up other tasks that are waiting on an item to appear in the queue. By default, data is stored in queues using **First In First Out** (**FIFO**) ordering – the first item to be put into the queue is the first item to be removed from the queue.

We'll start by looking at some simple behavior of queues when they are in different states and used in different ways (sending versus receiving), and then move on to how queues can be used to pass information between tasks.

Simple queue: sending an item

The first queue example is simply adding (also referred to as **sending**) an item to a queue that has empty space:

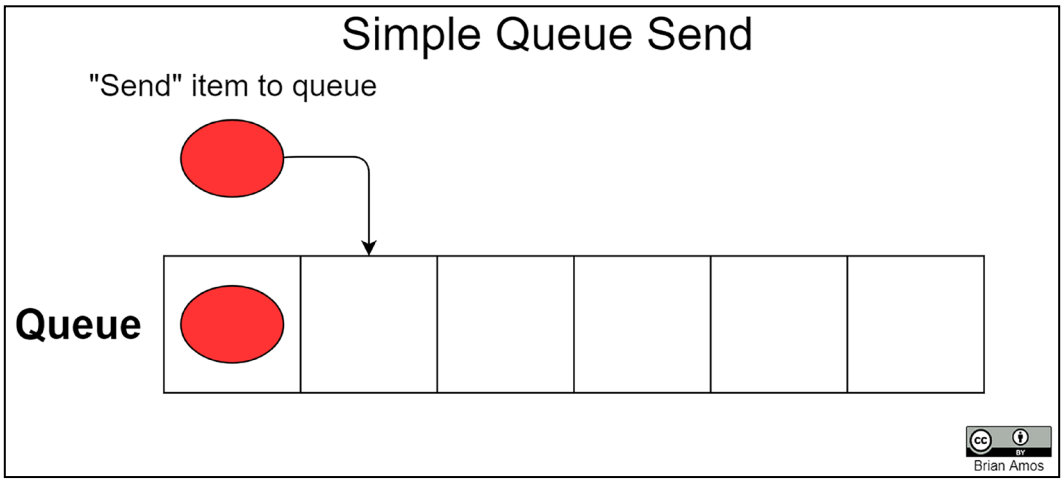


Figure 9.1 : Simple queue: sending an item

When an item is added to a queue with available space, the addition happens immediately. Because space was available in the queue, the task *sending* the item to the queue continues running, unless there is another higher-priority task waiting on an item to appear in the queue.

Although interaction with queues typically happens from within tasks, this isn't always the case. There are some special cases where queues can also be accessed from within ISRs (but that behavior has different rules). For the examples in this chapter, we'll assume that tasks are sending and receiving items from the queues.

Simple queue: receiving an item

In the following diagram, a task is shown **receiving** an item from a queue:

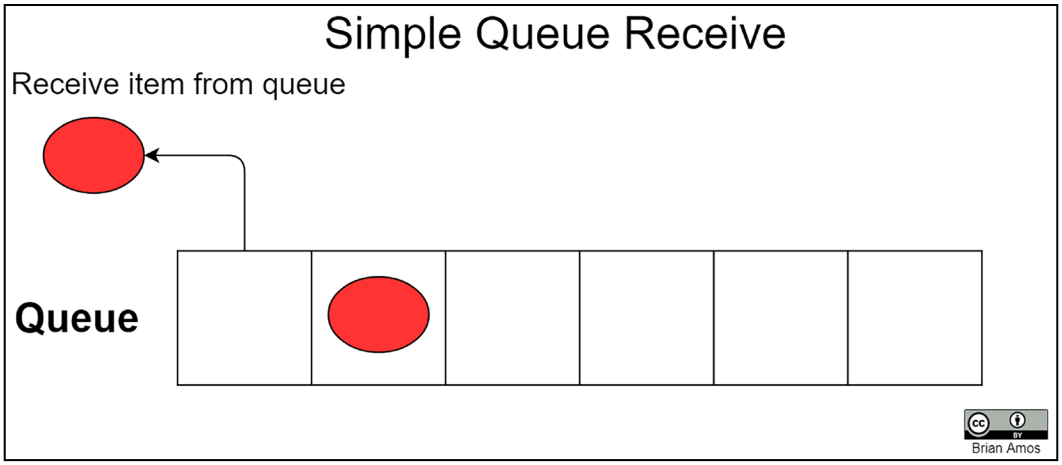


Figure 9.2 : Simple queue: receiving an item

When a task is ready to receive an item from a queue, it will get the oldest item, by default. In this example, since there is at least one item in the queue, the receive is processed immediately and the task continues to run.

Full queue: sending an item

When a queue is full, no information is thrown away. Instead, the task attempting to *send* the item to the queue will wait for up to a predetermined amount of time for available space in the queue:

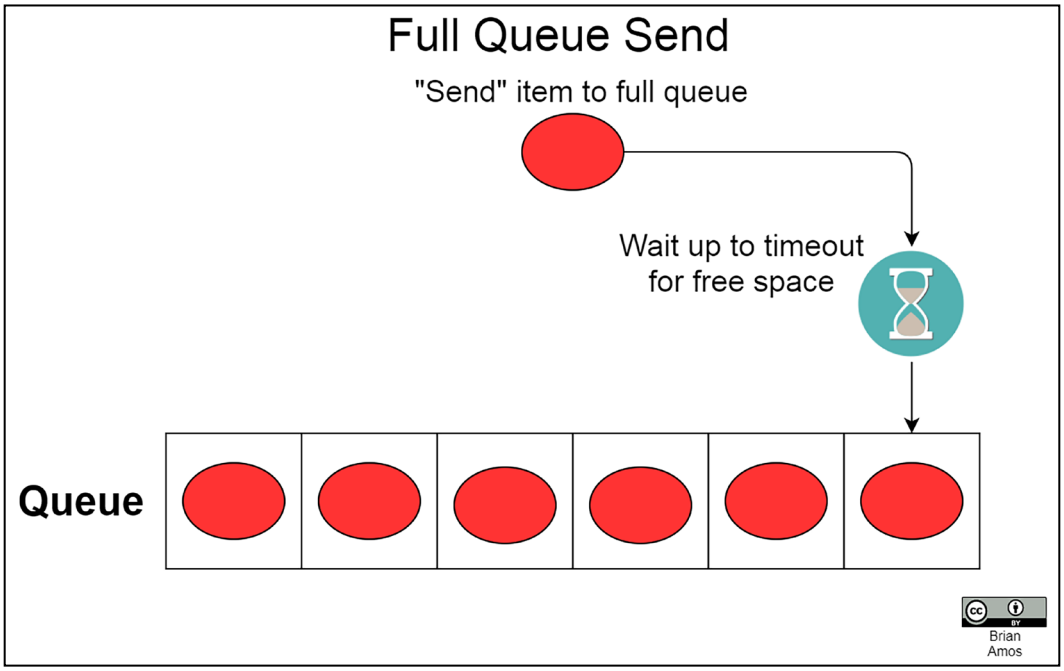


Figure 9.3 : Full queue: sending an item

When a queue is full, the task that is attempting to send an item to the queue will wait until a space in the queue becomes available, but only up to the timeout value specified.

In this example, if a task was attempting to send to a full queue, and it has a timeout of 10 ms, then it would only wait 10 ms for space to become available in the queue. After the timeout expires, the call will return and notify the calling code that the send has failed. What should be done about this failure is at the discretion of the programmer, and will vary depending on the use case. Extremely large timeout values can be used for truly non-critical functions. Just be aware that this will cause the sending task to effectively wait forever for a slot in the queue to become available (this is obviously no longer real time)!

It is up to you, as the programmer, to determine what an acceptable amount of time is on a case-by-case basis. It is also your responsibility to determine the severity and corrective actions if a timeout does occur. Potential corrective actions could range from nothing (think of a dropped frame in a video call) to an emergency shutdown.

Empty queue: receiving an item

Another case where accessing a queue can cause a task to block is when attempting to receive from an empty queue:

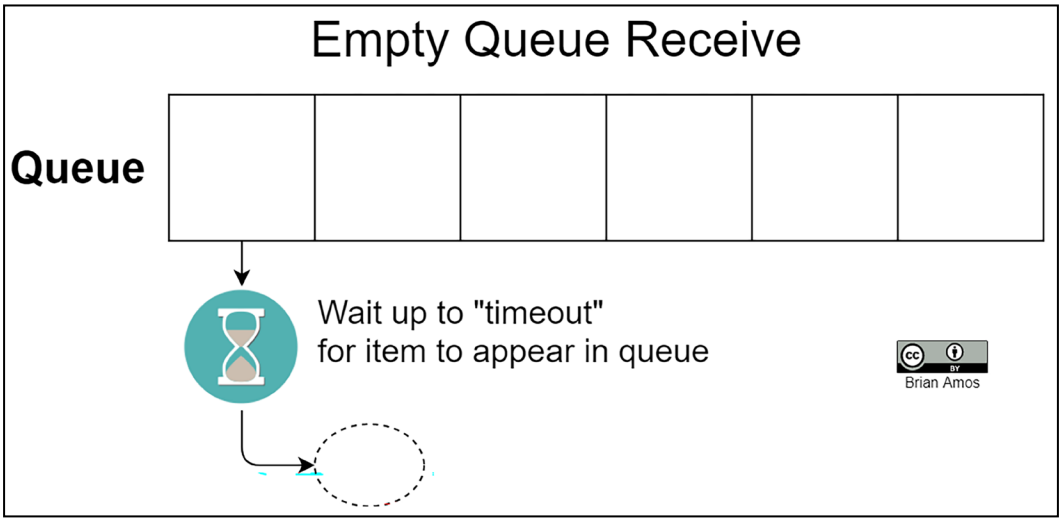


Figure 9.4 : Empty queue: receiving an item

Similar to a *send* waiting on space to become available, a task receiving from a queue also has the potential to be delayed. In the case of an empty queue, the task that is attempting to receive from the queue will be blocked until an item appears in the queue. If no item is available before the timeout expires, the calling code will be notified of the failure. Again, the exact course of action to take varies.

Sometimes, infinite waits are used. You'll often encounter very long wait periods for queues that are receiving input from external interfaces, as they may not be sending data constantly, e.g., serial ports. There is no issue at all if a human user on the other end of a serial port hasn't sent data for an extended period of time.

On the other hand, receive timeouts can also be used to ensure you have a minimum acceptable amount of data to process. Let's consider a sensor that supports readings at 10 Hz (10 readings per second). If you were implementing an algorithm that relies on fresh readings from this sensor, a timeout of slightly greater than 100 ms could be used to trigger an error. This timeout would guarantee that the algorithm is always acting on fresh sensor readings. In this case, hitting a timeout could be used to trigger a corrective action, or a notification that the sensor wasn't performing as expected.

Queues for intertask communication

Now that the simple behaviours of queues have been covered, we'll take a look at how queues can be used to move data between tasks. A very common use-case for queues is to have one task populate the queue while another task is reading from the same queue. This is generally straightforward, but it may have some nuances, depending on how the system is set up.

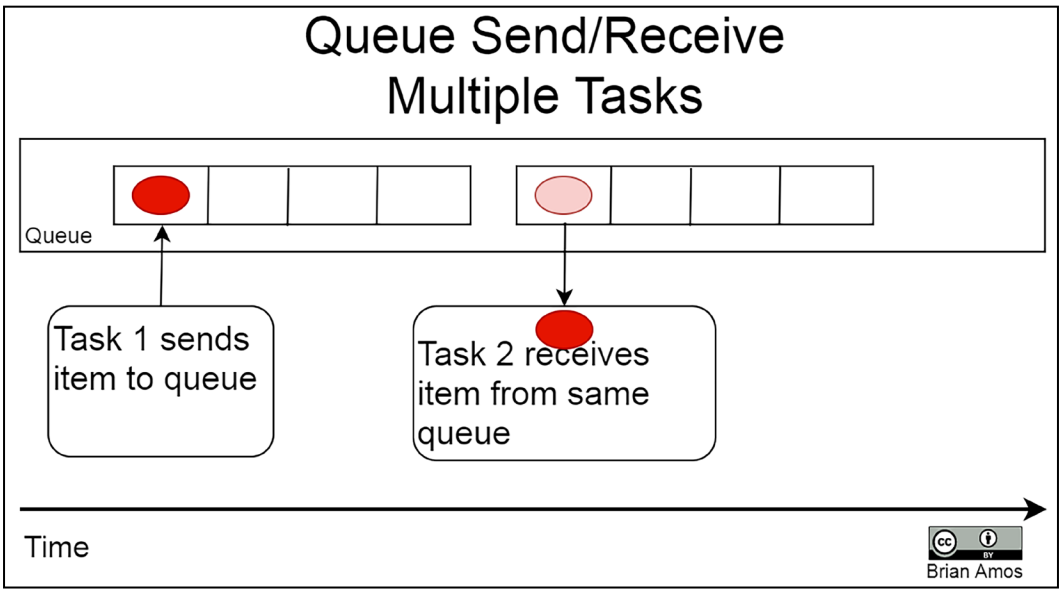


Figure 9.5 : When the task that receives items has the highest priority

In this example (shown above), the task that receives items has the highest priority. `Task1` sends items to the queue, and `Task2` receives items. `Task2` is blocked, waiting to receive an item from the empty queue. `Task1` sends an item to the queue. Since `Task2` has the highest priority, it will immediately receive the item. The queue is now empty again.

Let's consider another case that often occurs in practice: the task that sends items has the highest priority. If the highest-priority task always has data to send to the queue, the queue will fill before another task is given a chance to receive from the queue. Here is an example of how this may play out:

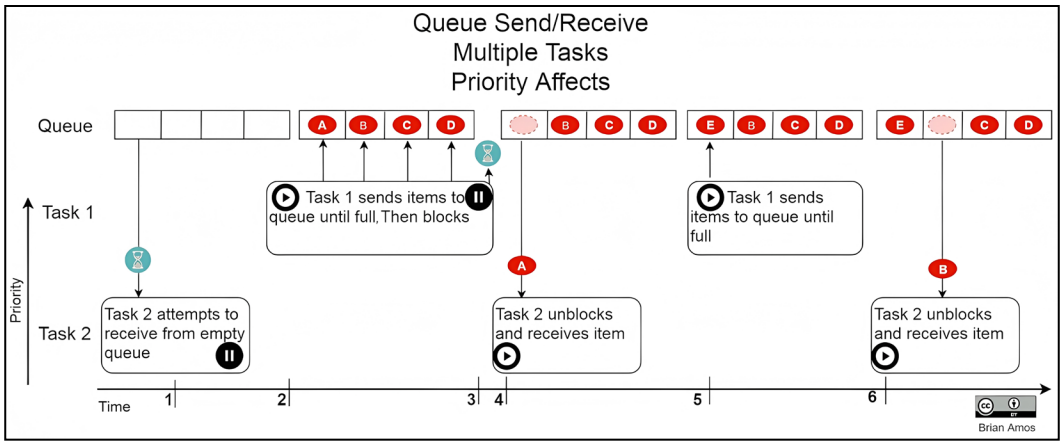


Figure 9.6 : When the task that sends items has the highest priority

The following numbers correspond to indexes along the time axis:

1. `Task2` attempts to receive an item from the empty queue. No items are available, so `Task2` blocks.
2. `Task1` sends items to the queue. Since it is the highest-priority task in the system, `Task1` sends items to the queue until it is full.
3. The queue is filled, so `Task1` is blocked.
4. `Task2` is given context by the scheduler since it is now the highest-priority task that may run. `Task2` receives the item from the queue. (However, `Task2` cannot process the item, as `Task1` is now unblocked. `Task2` can only process items when `Task1` is blocked or suspended.)
5. As soon as an item is removed from the queue, `Task1` is given context again. It is the highest-priority task in the system, and it is now able to run because it was blocking while waiting for space to become available in the queue. After `Task1` adds a single item, the queue is full and `Task1` is blocked.
6. `Task2` is given context and receives an item from the queue.

Another extremely common use-case for queues is to have a single queue accept input from many different sources. This is especially useful for something like a debug serial-port or a log file. Many different tasks can be **writing** (sending) to the queue. Also, there is a single task responsible for **reading** (receiving) data from the queue, and pushing the data out to a shared resource.

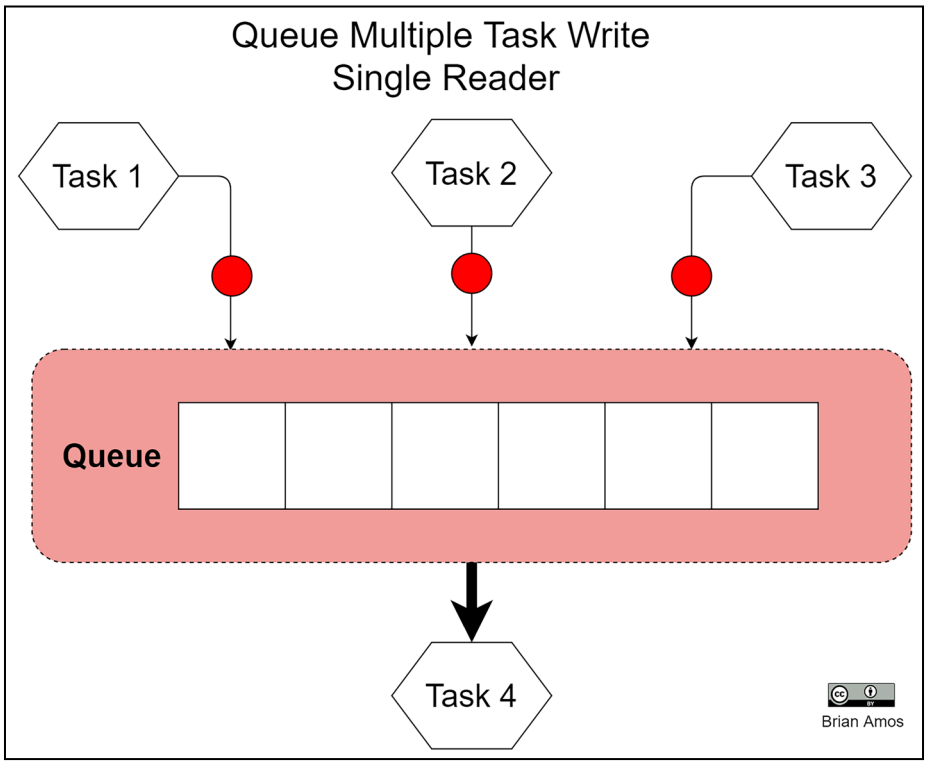


Figure 9.7 : A queue with multiple-tasks writing, and one reader

Next, we'll look at example-programs that illustrate these uses of queues, and that use FreeRTOS's APIs for queues.

Passing data through queues by value

Like semaphores and mutexes, queues are among the most widely used structures when using asynchronous tasks. They can be found in nearly every operating system, so it is beneficial to understand how to use them. We'll look at several different ways of using queues and interacting with them to affect a task's state.

In the following examples, we'll learn how to use queues as a means of sending *commands* to an LED state-machine. First, we'll examine a very simple use-case, passing a single one-byte value to a queue and operating on it.

Passing one byte by value

The first example-program we'll be using is `mainQueueSimplePassByValue.c`. It has two tasks, and they communicate with each other by using a queue. One task sends items to the queue (`sendingTask`), and the other task receives the items (`recvTask`). The queue's items are `uint8_t` values, and each item specifies which LEDs to turn on or off. When `recvTask` receives an item, it sets the LEDs on or off as specified, then waits 200 ms before attempting to receive another item.

So, let's begin. `main()` does the following:

1. Set up an `enum` for use in specifying the LED settings. A single LED's setting can be specified, or all LEDs'.

```
typedef enum
{
    ALL_OFF = 0,
    RED_ON = 1,
    RED_OFF = 2,
    BLUE_ON = 3,
    BLUE_OFF= 4,
    GREEN_ON = 5,
    GREEN_OFF = 6,
    ALL_ON = 7
} LED_CMDS;
```

2. Similar to the initialization paradigm of semaphores, queues must first be created and their handles stored so they can be used to access the queue later. Define a handle to point at the queue:

```
static QueueHandle_t ledCmdQueue = NULL;
```

3. Create the queue using the **xQueueCreate()** function:

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

Let's quickly outline what we see here:

- `uxQueueLength`: The maximum number of elements the queue can hold.
- `uxItemSize`: The size (in bytes) of each element in the queue.
- Return value: A handle to the queue that is created (or `NULL` upon error).

Our call to `xQueueCreate()` will look like the code below. Successful creation is verified before continuing.

```
ledCmdQueue = xQueueCreate(1, sizeof(uint8_t));
assert_param(ledCmdQueue != NULL);
```

Let's outline what we see here:

- The queue holds 1 element.
- The element is sized to hold `uint8_t`. A single byte is large enough to store the value of any enumeration we have explicitly defined.
- `xQueueCreate` returns a handle to the queue created, which is stored in `ledCmdQueue`. This "handle" is a global that will be used by the tasks when accessing the queue.

The beginning of our `recvTask()` looks like this:

```
void recvTask( void* NotUsed )
{
    uint8_t nextCmd = 0;

    while(1)
    {
        xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY);

        switch(nextCmd)
        {
            case ALL_OFF:
                RedLed.Off();
                GreenLed.Off();
                BlueLed.Off();
                break;
            case GREEN_ON:
                GreenLed.On();
                break;
            . . .
```

Let's have a close look at the actual queue-receive line highlighted in the preceding code:

```
xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY);
```

- The handle `ledCmdQueue` is used to access the queue.
- `nextCmd` is a local `uint8_t variable`, and defined on the stack. The address of this variable (a pointer) is passed. `xQueueReceive` will copy the next `LED_CMD` enumeration (stored as a byte in the queue) into `nextCmd`.
- **`portMAX_DELAY`** specifies an infinite timeout – that is, this function will never return if nothing is added to the queue (the same as timeouts for mutexes and semaphores).

The `sendingTask` is a simple `while` loop that sends different values of `LED_CMDS` to `ledCmdQueue`:

```
void sendingTask( void* NotUsed )
{
    uint8_t ledCmd;
    while(1)
    {
        for(int i = 0; i < 8; i++)
        {
            ledCmd = (LED_CMDS) i;
            xQueueSend(ledCmdQueue, &ledCmd, portMAX_DELAY);
            vTaskDelay(200/portTICK_PERIOD_MS);
        }
    }
}
```

The arguments for the sending side's `xQueueSend()` are nearly identical to the receiving side's `xQueueReceive()`, the only difference being that we're sending data *to* the queue this time:

```
xQueueSend(ledCmdQueue, &ledCmd, portMAX_DELAY);
```

- `ledCmdQueue`: The handle for the queue to send the data to.
- `&ledCmd`: The address of the data to send to the queue.
- `portMax_DELAY`: specifies an infinite timeout in waiting for the queue space to become available, if it is full.



Calls to `xQueueSend` can timeout if the queue remains full beyond the specified timeout, and the item isn't added to the queue. If your application has a non-infinite timeout (which in nearly all cases it should), you'll need to consider what should happen in this case. Courses of action could range from simply dropping the data item (it will be lost forever) to throwing an assert and going into some type of emergency state with a shutdown. A reboot is also popular in some contexts. The exact behavior will generally be dictated by the type of project or product you're working on.

When running the program on the dev-board, you'll notice that the LEDs follow the pattern defined by the `LED_CMDS` enum: `ALL_OFF`, `RED_ON`, `RED_OFF`, `BLUE_ON`, `BLUE_OFF`, `GREEN_ON`, `GREEN_OFF`, `ALL_ON`, with 200 ms between each transition.

But what if we decide we'd like to operate on more than one LED at a time? We *could* add more values to the existing `LED_CMDS` enum, such as `RED_ON_BLUE_ON_GREEN_OFF`, but that would be a lot of very error-prone typing, especially if we had more than 3 LEDs (8 LEDs results in 256 enum values to cover all combinations of each LED being on/off). Instead, let's look at how we can use a `struct` to describe the LED command and pass that through our queue.

Passing a composite data-type by value

The next example-program is `mainQueueCompositePassByValue.c`. It is very similar to the prior example-program. However, rather than using a single byte for the queue's item, a `struct` will be used.

In the FreeRTOS APIs for queues, the parameter for the queue item is of type `void*`. This is done to provide flexibility for the application writer. `void*` is a pointer to *anything,* and the queue-item's size is defined when the queue is created, so queues can be used to pass anything between tasks.



The use of `void*` for interacting with queues acts as a double-edged sword. It provides the ultimate amount of flexibility, but also provides the very real possibility for you to pass the *wrong* data-type into the queue, potentially without a warning from the compiler. You must keep track of the data-type that is being stored in each queue!

We'll take advantage of this flexibility in queue-item data-types, and use a **composite data-type** for the queue's items. It's a `struct`, made up of `uint8_t` bit-fields, each of which specifies an LED setting. The `struct` also contains a variable that specifies the delay-time for the LED settings.

```
typedef struct
{
    uint8_t redLEDState : 1;   // Specify this variable as 1 bit wide
    uint8_t blueLEDState : 1;  // Specify this variable as 1 bit wide
    uint8_t greenLEDState : 1; // Specify this variable as 1 bit wide
    uint32_t msDelayTime; // Minimum number of ms to remain in this state
} LedStates_t;
```

We'll also create a queue that is able to hold eight copies of the entire `LedStates_t` `struct`:

```
ledCmdQueue = xQueueCreate(8, sizeof(LedStates_t));
```

As in the prior example, the task `recvTask` waits until an item is available from the queue, and then it operates on the item (turning LEDs on/off as required):

```
void recvTask( void* NotUsed )
{
    LedStates_t nextCmd;

    while(1)
    {
        xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY);

        if(nextCmd.redLEDState == 1)
            RedLed.On();
        else
            RedLed.Off();
        if(nextCmd.blueLEDState == 1)
            BlueLed.On();
        else
            BlueLed.Off();
        if(nextCmd.greenLEDState == 1)
            GreenLed.On();
        else
            GreenLed.Off();

        vTaskDelay(nextCmd.msDelayTime/portTICK_PERIOD_MS);
    }
}
```

Here are the responsibilities of the primary loop of `recvTask`:

- Each time an element is available from the queue, each bit-field of the `struct` is evaluated and the appropriate action is taken. All three LEDs are updated with a single LED command, which is sent to the queue.
- The `msDelayTime` field is also evaluated (it is used to add a delay before the task attempts to receive from the queue again). This is what slows down the system enough so that the LED states are visible.

Here is the code from `sendingTask`:

```
void sendingTask( void* NotUsed )
{
    // A single instance of nextStates is defined here
    LedStates_t nextStates;

    while(1)
    {
        nextStates.redLEDState = 1;
        nextStates.greenLEDState = 1;
        nextStates.blueLEDState = 1;
        nextStates.msDelayTime = 500;
        xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY);

        nextStates.greenLEDState = 0;  // Turn-off the green LED
        xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY);

        nextStates.blueLEDState = 0;  // Turn-off the blue LED
        xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY);

        nextStates.redLEDState = 0;  // Turn-off the red LED
        nextStates.msDelayTime = 1000;
        xQueueSend(ledCmdQueue, &nextStates, portMAX_DELAY);
    }
}
```

The loop of `sendingTask` sends four commands to `ledCmdQueue` – here are the details:

- `sendingTask` looks a bit different from before. Now, since a `struct` is being passed, we can access each field, setting multiple fields before sending `nextStates` to the queue.
- Each time `xQueueSend()` is called, the contents of `nextStates` are copied into the queue before moving on. When `xQueueSend()` returns successfully, the value of `nextStates` has been copied into the queue storage; `nextStates` does not need to be preserved.

Internally, `xQueueSend` and `xQueueReceive` use `memcpy()` to copy data to and from the queue, respectively. Those APIs are defined in `Middleware\Third_Party\FreeRTOS\Source\queue.c`. The copies are performed by `prvCopyDataFromQueue()` and `prvCopyDataToQueue()`.

To drive home the point that the value of `nextStates` is copied into the queue, the example-program sets the tasks' priorities so that the queue is filled completely by `sendingTask` before having an item removed by `recvTask`. This is accomplished by giving `sendingTask` a higher priority than `recvTask`. Here's what our task definitions look like in `main()`. (`asserts` are present in the code but are not shown here to reduce clutter.):

```
xTaskCreate(recvTask, "recvTask", STACK_SIZE, NULL, (tskIDLE_PRIORITY + 1),
            NULL);
xTaskCreate(sendingTask, "sendingTask", STACK_SIZE, NULL,
            (configMAX_PRIORITIES - 1), NULL);
```



`sendingTask` is configured to have the highest possible priority in the system. `configMAX_PRIORITIES` is defined in `Chapter9/Inc/FreeRTOSConfig.h` and it specifies the number of priorities available. FreeRTOS's task priorities are defined such that `0` is the lowest possible priority, and the highest possible priority is `configMAX_PRIORITIES - 1`.

This prioritization setup allows `sendingTask` to repeatedly send data to the queue until it is full (because `sendingTask` has a higher priority). After the queue has filled, `sendingTask` will block and allow `recvTask` to remove an item from the queue. Let's take a look at how this plays out in more detail.

Understanding how queues affect execution

Task priorities work in conjunction with RTOS primitives, such as queues, to define the system's behavior. This is especially critical in a preemptive RTOS application because context is always given based on priority. Programmed queue-interactions need to take into account task priorities in order to achieve the desired operation. Priorities need to be carefully chosen to work in conjunction with the design of individual tasks.

In this example, an infinite wait time was chosen for `sendingTask` so that it could fill the queue. Here's a diagram depicting the preceding setup in action:

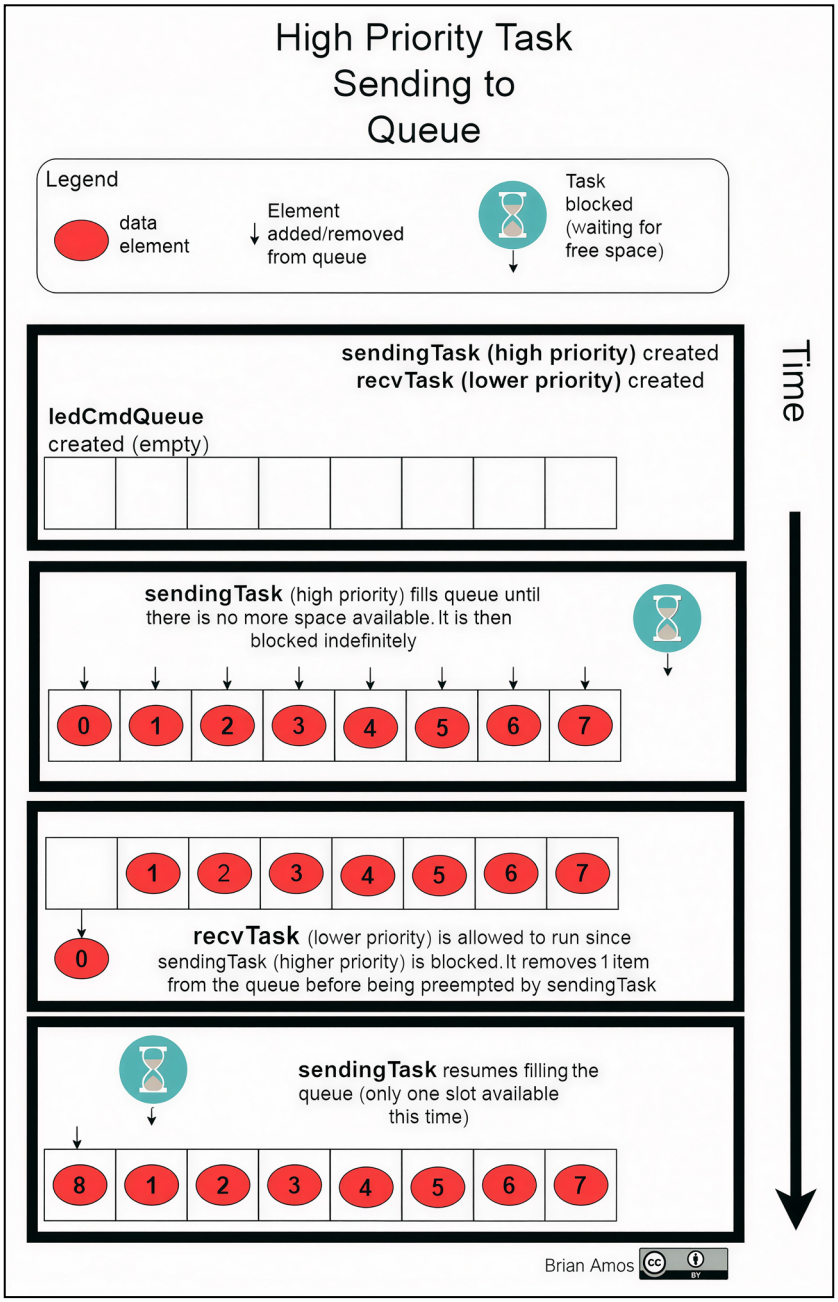


Figure 9.8 : The highest-priority task is sending to the queue

Take a look at this example using Ozone to step through the code and understand its behavior. We can go through a few iterations of `sendingTask's` `while`-loop, step by step. We'll watch the `ledCmdQueue` data-structure and breakpoint-setup in each of the tasks:

1. Build the configuration **queueCompositePassByValue**.
2. Set up the Ozone configuration-file to run the executable.
3. Open Ozone, using the configuration-file.
4. Open the **Global Data** view, to observe `ledCmdQueue` as you step through the code.
5. Put a breakpoint in `recvTask` to stop the debugger whenever an item is removed from the queue.
6. When `recvTask` runs the first time, you'll notice that `uxMessagesWaiting` will have a value of `8` (the queue is filled):

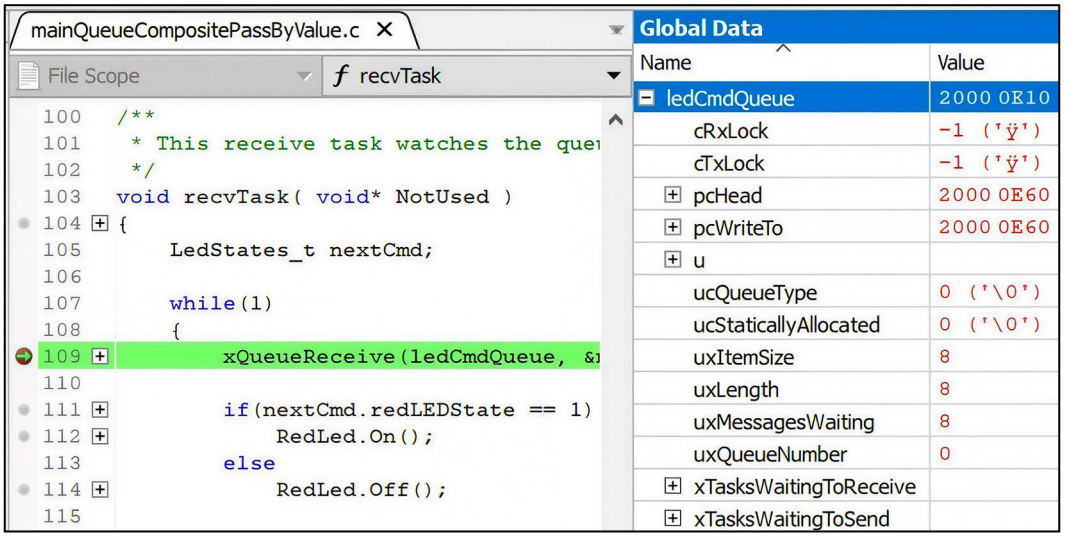


Figure 9.9 : Ozone, showing ledCmdQueue



Getting comfortable with whatever debugger you are using *before* you run into serious problems is always a good idea. A second-nature level of familiarity frees your mind to focus on the problem at hand rather than the tools being used.

Important notes on the example

The previous example's main purpose was to illustrate the following points:

- Queues can be used to hold arbitrary data.
- Queues interact with task priorities in interesting ways.

There were several trade-offs made to simplify behavior and make the example easier to understand:

- **The task receiving from the queue has the lowest priority**, and the sending task has the highest priority. This was done to show how a high-priority sender could potentially fill a queue. However, in this program, once the queue is full, when a queue-item is removed, another item is immediately added. The program would flash the LEDs in the same way if the queue just held one item.
- **A long queue was used for commands**: This example contains several seconds worth of queued commands. When an element is added to the queue, the command won't be executed until several seconds later. This was caused by having a long queue and by generating commands much faster than they are carried out.
- **An infinite timeout was used when sending items to the queue**: This will cause `sendTask()` to wait indefinitely for a slot to become available. This was the behavior we wanted here (for simplicity), but in an actual time-critical system, you'll need to keep in mind exactly how long a task is able to wait before an error occurs.

We're not quite done exploring the flexibility of queues. Next, we'll take a look at the case of passing data by reference to a queue.

Passing data through queues by reference

Since the data-type of a queue is arbitrary, we also have the ability to **pass data by reference** instead of by value. This works in a way that is similar to passing arguments by reference to a function.

The next example-program is in `mainQueueCompositePassByReference.c`. It is very similar to the prior example-program. However, instead of using queue items which are `structs`, it uses queue items that are pointers to `structs`.

When to pass by reference

A queue will make a copy of whatever data it is holding. So, if the data-structure being queued is large, it can be inefficient to pass the data-structure by value:

- Sending and receiving from queues forces a copy of the queue item each time.
- The resulting queue can get very large if the queue items are large data structures.

When large data structures need to be queued, passing the queue items by reference can be a good idea. Here's an example of a larger structure. After the compiler pads out this `struct`, it ends up being 264 bytes:

```
#define MAX_MSG_LEN 256
typedef struct
{
    uint32_t redLEDState : 1;
    uint32_t blueLEDState : 1;
    uint32_t greenLEDState : 1;

    uint32_t msDelayTime; // Minimum number of mS to remain in this state

    // An array for storing strings of up to 256 chars
    char message[MAX_MSG_LEN];
} LedStates_t;
```

If `structs` of type `LedStates_t` are used as queue items, 264 bytes must be copied every time an item is added to the queue, or removed from it. Instead, we can use queue items that are pointers to `structs` of type `LedStates_t`. Pointers are 4 bytes on Cortex-M processors. The instruction to create the queue is:

```
ledCmdQueue = xQueueCreate(8, sizeof(LedStates_t*));
```

Let's look at the difference between passing by value and passing by reference.

Passing by value

When passing by value, the data-use is as follows:

- `ledCmdQueue's` **size** would be about 2 KB (264 bytes * 8 items).
- 264 bytes are copied each time `xQueueSend()` or `xQueueReceive()` is called.
- When sending an item to the queue, the `struct` that is copied can be discarded immediately. (A full copy is present inside the queue.)

Passing by reference

When passing by reference, the data-use is as follows:

- `ledCmdQueue's` **size** would be 32 bytes (4 bytes * 8 items).
- A 4-byte pointer is copied each time `xQueueSend()` or `xQueueReceive()` is called.
- When sending an item to the queue, the `struct` that is pointed to cannot be discarded immediately. It is the only instance of that `struct` in the system. The `struct` must not be modified or freed until its pointer has been received from the queue, and its data is no longer needed.



With a queue, when passing by reference, we're making trade-offs between a) less processor use and less RAM consumption, and b) more complex code. The extra complexity comes from ensuring that the original value remains valid the entire time it is needed.

In the example-program, two instances of `LedStates_t` are used:

```
static const LedStates_t ledState1 = {1, 0, 0, 1000,
    "The quick brown fox jumped over the lazy dog. Only the Red LED is on."
};

static const LedStates_t ledState2 = {0, 1, 0, 1000,
    "Another string. Only the Blue LED is on"};
```

Using Ozone, we can easily look at what we've created:

1. `uxItemSize` of `ledCmdQeue` is 4 bytes, exactly as we would expect, because the queue is holding pointers to `LedStates_t`.
2. The actual sizes of `ledState1` and `ledState2` are both 264 bytes, as expected:

Global Data

Name	Value	Location	Size
led*	*	*	*
ledCmdQueue	2000 0E00	2000 5118	4
pcHead	2000 0E50 "┌"	2000 0E00	4
pcWriteTo	2000 0E54 "┐"	2000 0E04	4
u		2000 0E08	8
xTasksWaitingToSend		2000 0E10	20
xTasksWaitingToReceive		2000 0E24	20
uxMessagesWaiting	8	2000 0E38	4
uxLength	8	2000 0E3C	4
uxItemSize	4	2000 0E40	4
cRxLock	-1 ('ÿ')	2000 0E44	1
cTxLock	-1 ('ÿ')	2000 0E45	1
ucStaticallyAllocated	0 ('\0')	2000 0E46	1
uxQueueNumber	0	2000 0E48	4
ucQueueType	0 ('\0')	2000 0E4C	1
ledState1		2000 0024	264
redLEDState	1	2000 0024	1
blueLEDState	0	2000 0024	1
greenLEDState	0	2000 0024	1
msDelayTime	1 000	2000 0028	4
message	"The quick brown	2000 002C	256
ledState2		2000 012C	264
redLEDState	0	2000 012C	1
blueLEDState	1	2000 012C	1
greenLEDState	0	2000 012C	1
msDelayTime	1 000	2000 0130	4
message	"Another string.	2000 0134	256

Figure 9.10 : Ozone, showing the queue-item's size

To send an item to the queue, go through the following steps:

1. Create a pointer to the variable (it's actually a `const`, here). And, when calling `xQueueSend`, pass in the address to the pointer:

```
void sendingTask( void* NotUsed )
{
    const LedStates_t * state1Ptr = &ledState1;
    const LedStates_t * state2Ptr = &ledState2;

    while(1)
    {
        xQueueSend(ledCmdQueue, &state1Ptr, portMAX_DELAY);
        xQueueSend(ledCmdQueue, &state2Ptr, portMAX_DELAY);
    }
}
```

2. To receive items from the queue, simply define a pointer of the correct data type. And, when calling `xQueueReceive`, pass the address to the pointer:

```
void recvTask( void* NotUsed )
{
    LedStates_t* nextCmd;

    while(1)
    {
        xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY);

        if(nextCmd->redLEDState == 1)
            RedLed.On();
        else
            RedLed.Off();
        . . .
        vTaskDelay(nextCmd->msDelayTime/portTICK_PERIOD_MS);
    }
}
```

When operating on an item taken out of the queue, remember that you've got a pointer that needs to be dereferenced (that is, `nextCmd->redLEDState`).

In this example-program, `recvTask` has higher priority than `sendingTask`. So, `sendingTask` will keep the queue full during the delay in `recvTask`.

Now for the catches...

Important notes

Passing by reference can be more efficient than passing by value for moving large data-structures around, but several things need to be kept in mind:

- **Keep the data types straight**: Because the argument to a queue is of the `void*` data type, the compiler won't be able to warn you that you're supplying an address to a `struct` instead of to a pointer.
- **Keep the queued data around**: Unlike passing data by value, when a queue holds pointers to the data, the underlying data needs to be available and unaltered until it is used. A stable storage location is needed. Global variables can be used. Alternatively, space for the variable can be dynamically allocated, if dynamic allocation is acceptable in your application. (Dynamic allocation is discussed in *Chapter 14, FreeRTOS Memory Management*.)
- **Who owns the data?** When a queue has a copy of a `struct`, the queue owns that copy. As soon as the item is removed from the queue, it disappears. Contrast this with a queue holding a *pointer* to data. When the pointer is removed from the queue, the data is still present in its previous location. Data ownership needs to be made very clear. Will the task receiving the pointer from the queue become the new owner (and be responsible for freeing dynamically allocated memory if it was used)? Will the original task that sent the pointer still maintain ownership? These are all important questions to consider upfront.

In the example-program, the queue items point to data structures that are declared as global constants. Since the data structures cannot change, we do not need to define which task owns them.

We've discussed passing around huge amounts of data, and how to avoid copying it to and from the queue. Next, let's talk about an efficient way of passing around small amounts of data.

Task notifications

Queues are an excellent workhorse of an RTOS because of their flexibility. Sometimes, all of this flexibility isn't needed and we'd prefer a more lightweight alternative for tasks to communicate with each other. FreeRTOS provides this with its **task notifications feature.**

A task notification has two main components: the notification itself, and a 32-bit value. A task can send a notification to another task. Conversely, a task can read a notification that has been sent to it. Each task has a **notification state**, which can be **pending** or **not-pending**. Each task also has a **notification value**, which is a 32-bit unsigned integer. The notification value is stored in the task's **Task Control Block (TCB)**. The TCB is defined in `Middleware\Third_Party\FreeRTOS\Source\tasks.c`.

When a notification is sent to a task, its notification state is set to pending. When a task reads its *notification value*, its notification state is set to *not-pending*. When a task attempts to read its notification value, but its notification state is not-pending, the task can enter the Blocked state, and wait for its notification state to become *pending*. A timeout can be specified for the wait.

Let's see an example of how task notifications can be used as a substitute for a queue. In the next example-program, we'll use task notifications to issue commands and pass information to a task.



If you're thinking *task notifications also sound a bit like semaphores*, you'd be right! Task notifications can also be used as a faster alternative to semaphores.

Passing simple data using task notifications

The next example-program is `mainTaskNotifications.c`, and it is similar to the prior example-programs. The task `recvTask` receives commands to set LEDs. The task `sendingTask` sends the commands. `recvTask` has higher priority than `sendingTask`.

Since the notification value is built into the task, no additional queue needs to be created. We just need to make sure that we store the task handle of `recvTask`, which will be used when we send it notifications.

To see how this is done, let's look at some excerpts from `mainTaskNotifications.c`.

Outside of `main`, we'll define some bitmasks and a task handle:

```
#define RED_LED_MASK   0x00000001
#define BLUE_LED_MASK  0x00000002
#define GREEN_LED_MASK 0x00000004

static xTaskHandle recvTaskHandle = NULL;
```

Inside `main`, we'll call `xTaskCreate` to create `recvTask`, and pass it the handle to populate:

```
retVal = xTaskCreate(recvTask, "recvTask", STACK_SIZE, NULL,
                     tskIDLE_PRIORITY + 2, &recvTaskHandle);
assert_param( retVal == pdPASS);
```

`recvTask` waits on the next notification by calling `xTaskNotifyWait`. After a notification is received, the task evaluates each LED's mask, and turns LEDs on or off, accordingly. For `xTaskNotifyWait`, its first parameter specifies the bits to clear in the task's notification value, on entry to the function. The second parameter specifies the bits to clear on exit from the function. In our call to `xTaskNotifyWait`, the first two arguments are 0, specifying no bits should be cleared.

```
void recvTask( void* NotUsed )
{
    uint32_t notificationvalue;

    while(1)
    {
        // Wait for the next notification value
        xTaskNotifyWait(0, 0, &notificationvalue, portMAX_DELAY);

        if((notificationvalue & RED_LED_MASK) != 0)
            RedLed.On();
        else
            RedLed.Off();
```

`sendingTask` calls `xTaskNotify` to send a notification. `xTaskNotify`'s third parameter specifies the **action** to be taken. The argument `eSetValueWithOverwrite` specifies to overwrite the notification value in the TCB. For this action, `xTaskNotify` always returns `pdTRUE`:

```
void sendingTask( void* NotUsed )
{
    while(1)
    {
        xTaskNotify( recvTaskHandle, RED_LED_MASK, eSetValueWithOverwrite);
        vTaskDelay(500);
```

The example-program can be built using the configuration **directTaskNotification,** and it can be run on the dev-board. The program will sequentially blink each LED as the notifications are sent to `recvTask`.

Other actions for task notifications

The other actions that can be taken for task notifications are listed in the `eNotifyAction` enumeration in `task.h`. (Of course, they are also listed in the FreeRTOS reference manual.)

```
eNoAction = 0,   /* Notify the task without updating its notify value.  */
eSetBits,        /* Set bits in the task's notification value.          */
eIncrement,      /* Increment the task's notification value.            */
eSetValueWithOverwrite,    /* Set the task's notification value to a specific
                              value even if the previous value has not yet
                              been read by the task.                    */
eSetValueWithoutOverwrite  /* Set the task's notification value if the
                               previous value has been read by the task. */
```

Using these actions creates some additional flexibility, such as using notifications as binary and counting semaphores. Note that some of these options change how `xTaskNotify` returns, so it will be necessary to check the return value in some cases.

Comparing task notifications to queues

Task notifications are similar to the other communication mechanisms we've discussed, e.g., queues and semaphores. However, task notifications do not require that a communication-object be instantiated in RAM, as is required with queues and semaphores. Also, task notifications use fewer CPU cycles than semaphores and queues, so task notifications are faster (between 35% and 45% faster).

Compared to queues, task notifications have the following features:

- Task notifications have storage capacity of exactly one 32-bit integer.
- When `xTaskNotify` is called, and the receiving task's notification-state is pending, the sending task does not enter the Blocked state. That is, the sending task does not block to wait for the receiving-task's notification state to change to not-pending. Instead, if the `xTaskNotify` action is `eSetValueWithOverwrite`, the receiving task's notification value is overwritten. If the action is `eSetValueWithoutOverwrite`, then `xTaskNotify` will return immediately, without writing to the receiving task's notification-value.
- Only one task can be notified at a time (since the notification value is stored inside the receiving task).

Let's take a look at a real-world example using SystemView to compare our task-notification code with the first queue implementation.

The queue implementation from `mainQueueSimplePassByValue.c` looks like this when performing `xQueueSend`:

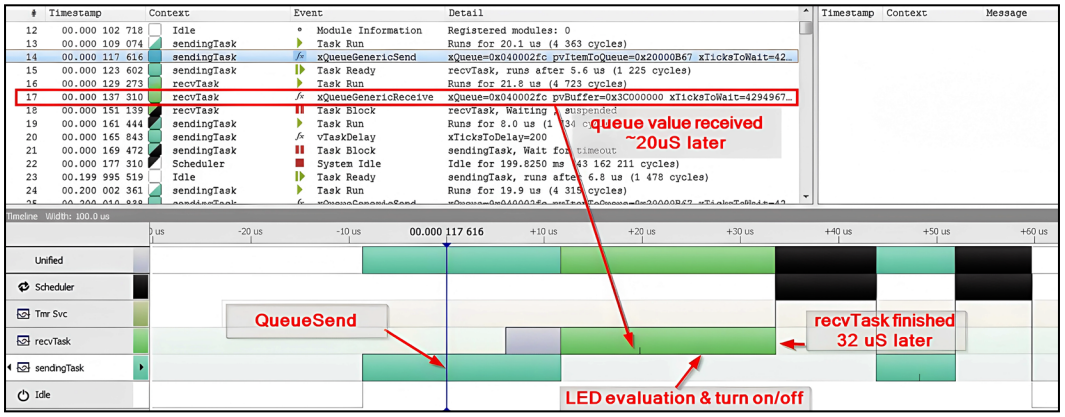


Figure 9.11 : Using SystemView to visualize xQueueSend

Task-notification looks like this when calling `xTaskNotify` (you'll probably need to turn-off **Show ISRs**):

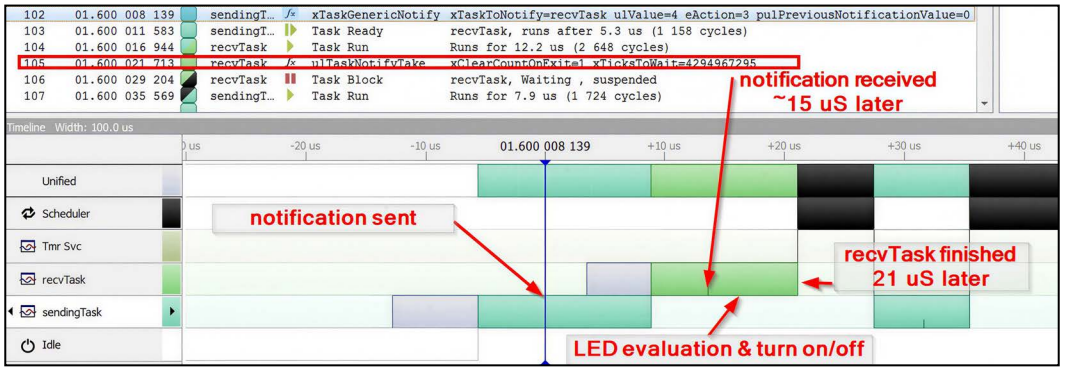


Figure 9.12 : Using SystemView to visualize xTaskNotify

As we can see from the preceding screenshots, the task-notify was around 33% faster than sending to the queue (20µs / 15µs). There's also no RAM overhead for storing a queue when using task notifications.

Indexed task-notifications

With `xTaskNotify`, each task has a single notification value. Starting in FreeRTOS V10.4.0, tasks can have an array of notification values. And, notification arrays can be enabled in `FreeRTOSConfig.h`. The APIs for using notification-arrays are `xTaskNotifyIndexed` and `xTaskNotifyGiveIndexed`. More information can be found at `https://www.freertos.org/Documentation/02-Kernel/04-API-references/05-Direct-to-task-notifications/00-RTOS-task-notifications`.

Among the example-programs, a notification array can be used with the IDE project `FreeRTOS_project`. As of the writing of this book, for the other example-programs, their FreeRTOS version doesn't support notification arrays. If newer FreeRTOS versions are provided in the future, it will be specified on the GitHub repository.

Summary

You've now learned what FreeRTOS queues are, and the basics of how to use them in a variety of scenarios, such as passing simple and composite elements, and passing by value and reference. You're aware of the pros and cons of using queues to store references to objects and when it is appropriate to use this method. We also covered some of the detailed interactions between queues, tasks, and task priorities. We finished with a simple real-world example of how to use task notifications to efficiently drive a small state machine.

As you become more accustomed to using RTOSes to solve a wide variety of problems, you'll find new and creative ways of using queues and task notifications. Tasks, queues, semaphores, and mutexes are truly the building blocks of RTOS-based applications and will help you go a long way.

We're not completely done with any of these elements yet, though – there's still a lot of more advanced material to cover related to using all of these primitives in the context of ISRs, which is up next!

Further reading

- The FreeRTOS configuration-settings in `FreeRTOSConfig.h`: `https://www.freertos.org/a00110.html`
- Task notifications:
    - *Mastering the FreeRTOS™ Real Time Kernel*, `https://www.freertos.org/Documentation/RTOS_book.html`
    - *RTOS Task Notifications*, `https://www.freertos.org/RTOS-task-notifications.html`

Part 3

Using FreeRTOS to Build Systems

In Part 3 of the book, we'll show how FreeRTOS can be used to construct drivers for the MCU's UARTs and USB controller. We'll use FreeRTOS tasks, semaphores, mutexes, and queues. An additional FreeRTOS feature is presented for sending data between tasks – stream buffers. The drivers' tasks interact with interrupt service routines (ISRs) and direct memory access (DMA). Short tutorials are provided for the hardware used. By the end of Part 3, you'll have a good understanding of how FreeRTOS can be used to develop real systems. You will get hands-on experience with the dev-board and example programs.

Part 3 includes the following chapters:

- *Chapter 10, Drivers and ISRs*
- *Chapter 11, More Efficient Drivers and ISRs*
- *Chapter 12, Sharing Hardware Peripherals among Tasks*
- *Chapter 13, Creating Loose Coupling with Queues*

10

Drivers and ISRs

Interacting with MCU peripherals is extremely important in many applications. In this chapter, we'll discuss several ways of implementing **drivers** for MCU peripherals. Up to this point, we've been using blinking LEDs as a means of interacting with our dev-board. This is about to change. We'll now focus on creating drivers for **UART**s (**universal asynchronous receiver/transmitter)**. UARTs are a widely-used communication peripheral.

We'll start by exploring a UART peripheral on our MCU, and we'll implement an extremely simple driver for it inside a task. The driver uses polling, and it is receive-only. After taking a look at the driver's performance limitations, we'll look at **interrupt service routines (ISRs)** and the different ways ISRs can interact with the RTOS kernel. The driver will then be implemented using interrupts and an ISR. Throughout this chapter, we'll use SystemView to keep a close eye on system performance.

The following topics will be covered in this chapter:

- Introducing the UART
- Creating a polled UART driver
- FreeRTOS tasks vs. ISRs
- Creating ISR-based drivers:
    - A queue-based driver
    - A buffer-based driver

The chapter's example-programs are in the IDE-project `Chapter_10`. There are three example-programs, and each has its own build-configuration. Two jumper wires are needed for the dev-board. Pre-terminated jumper wires can be used, or 20-22 AWG (~0.65 mm) solid-core wires.

For the reference-manuals cited in this chapter, complete bibliographic information can be found in *Appendix B: Reference Information*.

Introducing the UART

The chapter's example-programs read data from a UART on the MCU. The MCU has multiple UARTs. The one that we'll read from is USART2. To send data to USART2, we'll set up another UART on the MCU. It is UART4. Data is continuously generated for UART4, which it sends to USART2. The generated data is the string "`data from uart4`," and UART4 sends it repeatedly, at 9600 baud.

This section will further describe the UARTs, including how they are configured and started.

UART hardware takes bytes of data and transmits them serially over a wire by modulating the voltage of a signal-line at a predetermined rate:

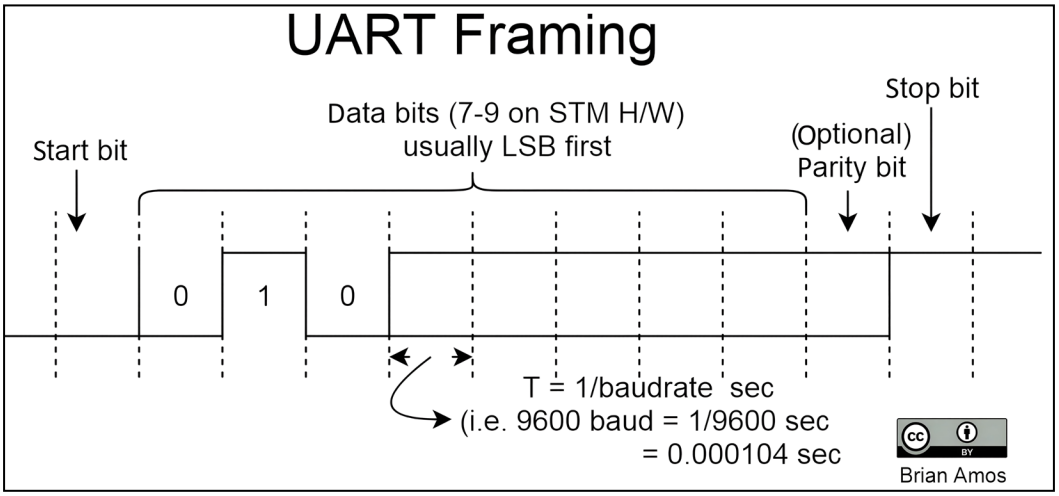


Figure 10.1 : Framing of the UART data packet

The *asynchronous* nature of a UART means no additional clock line is needed to monitor individual bit transitions. Instead, the hardware is set up to transition each bit at a specific frequency (**baud rate**). The UART hardware also adds some extra framing to the beginning and end of each packet it transmits. Start and stop bits signal the beginning and end of a packet. These bits (along with an optional parity bit) are used by the hardware to help guarantee the validity of packets (which are typically 8 bits long).



A more general form of UART hardware is the *USART*, the u**niversal synchronous/asynchronous receiver/transmitter**. USARTs are capable of transferring data either synchronously (with the addition of a clock signal) or asynchronously (without a clock signal).

The term *UART* can also be used to refer to UART-type devices, i.e., UARTs and USARTs, when the context makes it clear.

UARTs are often used to communicate between different chips and systems. They form the foundation of many different communication solutions, such as RS232, RS422, RS485, Modbus, and so on. UARTs can also be used for multi-processor communication and to communicate with different chips in the same system – for example, Wi-Fi and Bluetooth transceivers.



The UART is a common peripheral on many MCUs and has a long history in electronics. While there are many peripherals that are common today that offer higher speeds (e.g., CAN, USB, and Ethernet), the humble UART still has many uses and is worth understanding.

Our MCU has several UART peripherals. We'll use two of them, and configure them to communicate with each other. They are UART4 and USART2. They will communicate over jumper-wires that we'll install on the dev-board (shown in *Figure 10.2* below).

The wiring is shown in the diagram below. The transmit signal (Tx) from UART4 is connected to the receive signal (Rx) of USART2. Also, the transmit signal from USART2 is connected to the receive signal of UART4. The two wires provide bidirectional communication between the two UARTs.

On the dev-board, make the connections shown in the diagram below. Pre-terminated jumper wires can be used, or 20-22 AWG (~0.65 mm) solid-core wires. The dev-board's user-manual specifies its connectors and pins. (The dev-board's documentation is listed in *Appendix B, Reference Information*.)

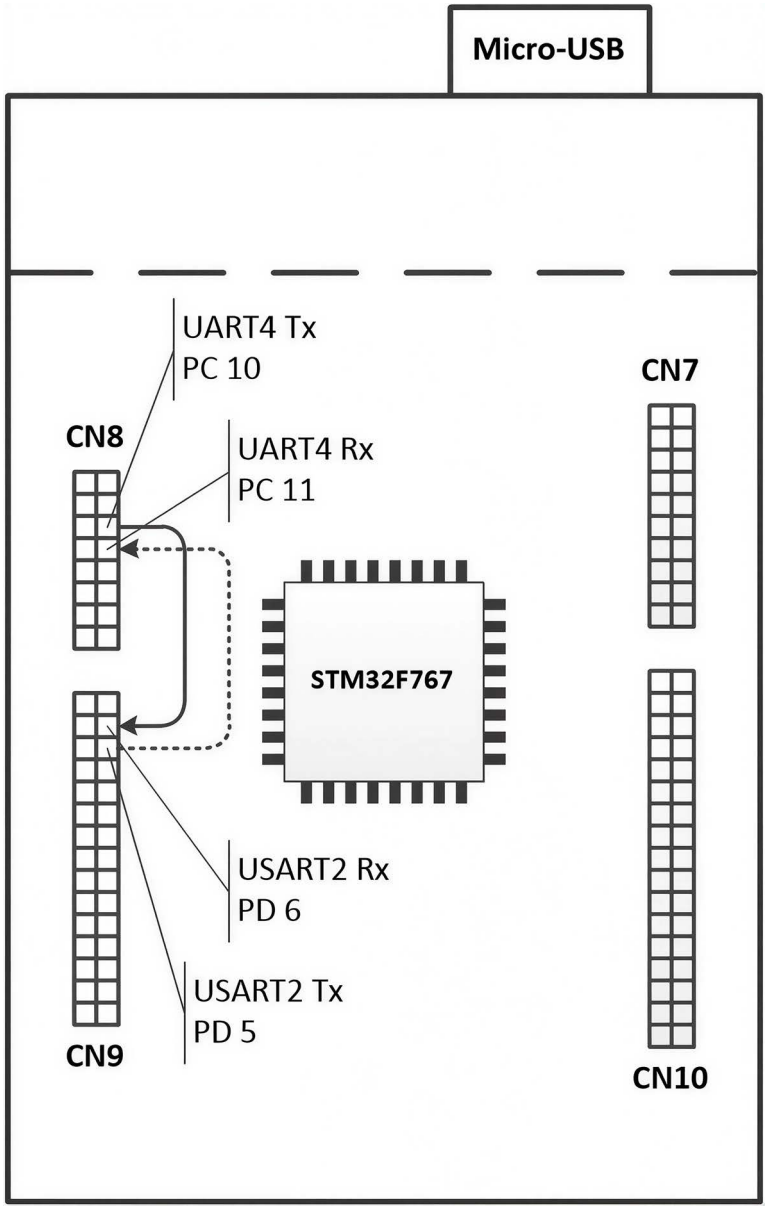


Figure 10.2 : The jumper-wires used to connect the two UARTs

Now that the connections are made, let's take a closer look at what else needs to happen before we can consider transferring data between peripherals on this MCU.

Setting up the UART

As we can see from the following simplified block diagram, there are a few components involved when setting up a UART for communication. The UART needs to be properly configured to transmit at the correct baud-rate, parity settings, flow control, and stop bits. Other hardware that interacts with the UART will also need to be configured, e.g., the GPIO pins.

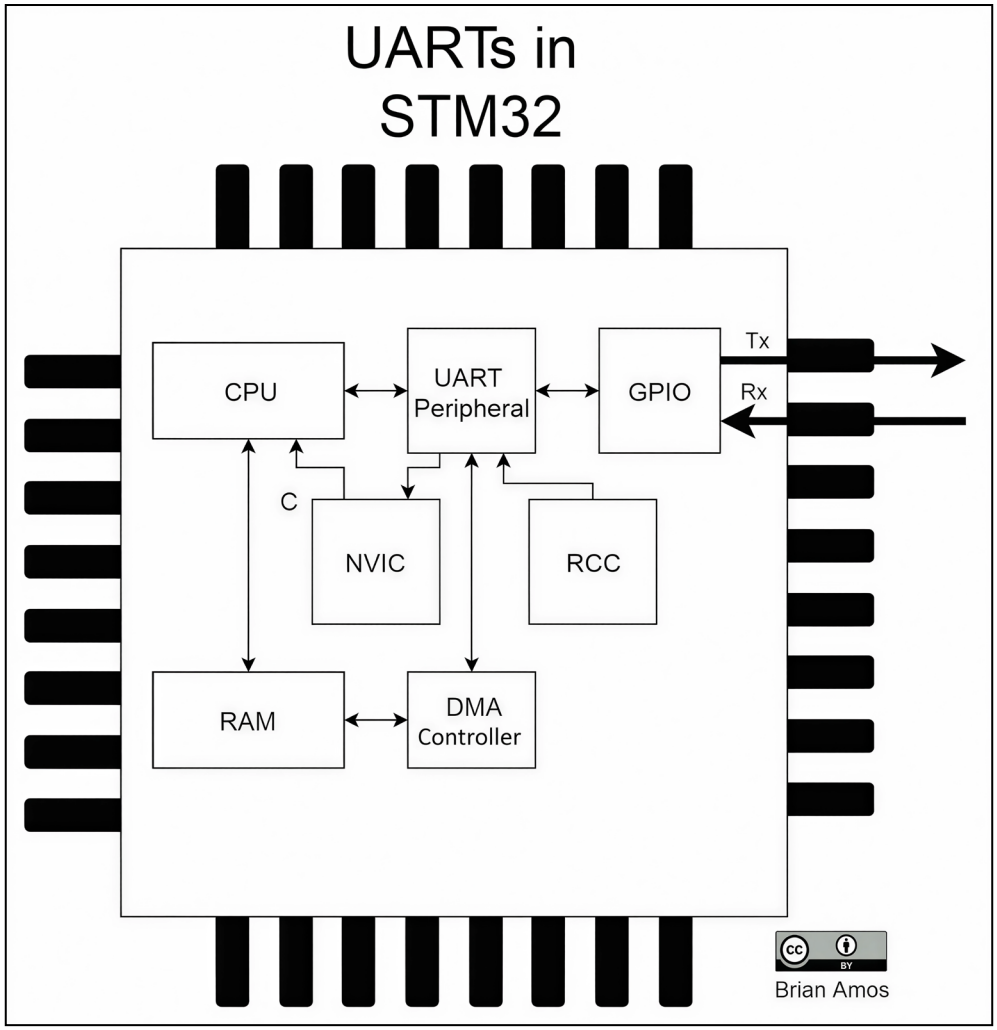


Figure 10.3 : Simplified block diagram of the STM32 UART

In the chapter's example-programs, the two UARTs are set up by calling functions in `BSP/UartQuickDirtyInit.c`. The functions are described here.

The steps for setting up UART4 are listed below. (This approach is used for most peripherals that attach to MCU pins.)

1. Configure the **GPIO** lines. (GPIO is described in *Chapter 5, Implementing the Super-Loop*.) Since each GPIO pin on the MCU can be shared with many different peripherals, the GPIO pins must be configured for the desired peripheral (in this case, UART4). In this example, we'll cover the steps to configure GPIO pins PC10 and PC11 as the UART4 transmit and receive signals, respectively.



You can read more about the MCU's pinouts in its datasheet, in *Section 3, Pinouts and Pin Description*. (The datasheet's bibliograhic information is in *Appendix B*.)

Datasheets will typically contain information specific to exact models of MCUs, while reference manuals will contain general information about an entire family of MCUs. The following excerpt is from the datasheet, and it shows alternate-function pin-mappings:

Table 13. STM32F765xx, STM32F767xx, STM32F768Ax and STM32F769xx alternate function mapping (continued)

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF1
		SYS	I2C4/UART5/TIM1/2	TIM3/4/5	TIM8/9/10/11/LPTIM1/DFSDM1/CEC	I2C1/2/3/4/USART1/CEC	SPI1/I2S1/SPI2/I2S2/SPI3/I2S3/SPI4/5/6	SPI2/I2S2/SPI3/I2S3/SAI1/I2C4/UART4/DFSDM1	SPI[illegible]/USART1/2/3/UART5/DFSDM1/SPDIF	SPI6/SAI2/USART6/UART4/5/7/8/OTG_FS/SPDIF	CAN1/2/TIM12/13/14/QUADSPI/FMC/LCD	SAI2/QUADSPI/SDMMC2/DFSDM1/OTG2_HS/OTG1_FS/LCD	I2C4/CAN3/SDMMC2/ETH	UAR FMC/ MMC DIOS G2_
Port C	PC11	-	-	-	DFSDM1_DATAIN5	-	-	SPI3_MISO	USART3_RX	UART4_RX	QUADSPI_BK2_NCS	-	-	SDM _D
	PC12	TRACED3	-	-	-	-	-	SPI3_MOSI/I2S3_SD	U[illegible]	UART5_TX	-	-	-	SDM _C
	PC13	-	-	-	-	-	-	-	-	-	-	-	-	-

Figure 10.4 : The datasheet's GPIO pin specifications

2. Reference the desired port and bit. In this case, we'll set up Port C and bit 11 (`PC11`), to map to the `UART4_Rx` function.
3. Find the desired alternate-function for the pin: `UART4_Rx`.
4. Find the alternate-function number to use when configuring the GPIO registers: `AF8`.
5. Set up the appropriate GPIO registers to correctly configure the hardware and map the desired UART peripheral to the physical pins:

   The following code is an excerpt from the function `initUart4Pins()`. It initializes the pins for UART4.

   ```
   GPIO_InitTypeDef GPIO_InitStruct = {0};

   // PC10 is UART4_TX, PC11 is UART4_RX
   GPIO_InitStruct.Pin = GPIO_PIN_10 | GPIO_PIN_11;
   GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
   GPIO_InitStruct.Pull = GPIO_NOPULL;
   GPIO_InitStruct.Alternate = GPIO_AF8_UART4;
   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
   ```

   For simplicity, the GPIO pins are set up by using `HAL_GPIO_Init`, an STM-supplied HAL function. The appropriate GPIO registers will be written when `HAL_GPIO_Init` is called. All we need to do is fill in a `GPIO_InitTypeDef struct`, and pass the `struct` by reference to `HAL_GPIO_Init`. In the code, the GPIO pins 10 and 11 are both initialized to alternative push/pull functions. They are also mapped to UART4 by setting the alternate-function variable to `AF8` – which was determined in step 4. When `HAL_GPIO_Init` is called, `GPIOC` specifies to use Port C.

6. Enable the necessary peripheral clocks. Since each peripheral clock is turned off by default (for power saving), the UART's peripheral clock must be turned on by writing to the **RCC** register (**Reset and Clock Control**). The following line is in the function `STM_UartInit`. It is from the HAL:

   ```
   __UART4_CLK_ENABLE();
   ```

7. Configure the interrupts (if using them) by configuring settings in the **nested vector interrupt controller** (**NVIC**) – details will be included for the example-programs where appropriate.

8. Configure the Direct Mssemory Access (DMA) (if using it) – details will be included in the examples where appropriate.
9. Configure the peripheral with the necessary settings, such as baud rate, parity, flow control, and so on.

   The following code is an excerpt from the function `STM_UartInit`. The function's input parameters include the baud-rate (`Baudrate`) and the UART being initialized (`STM_UART_PERIPH`). `STM_UartInit` makes it very easy to configure multiple UART peripherals with similar settings, without repeating all of the following code every time:

```
HAL_StatusTypeDef retVal;
UART_HandleTypeDef uartInitStruct;
. . .
uartInitStruct.Instance = STM_UART_PERIPH;
uartInitStruct.Init.BaudRate = Baudrate;
uartInitStruct.Init.WordLength = UART_WORDLENGTH_8B;
uartInitStruct.Init.StopBits = UART_STOPBITS_1;
uartInitStruct.Init.Parity = UART_PARITY_NONE;
uartInitStruct.Init.Mode = UART_MODE_TX_RX;
uartInitStruct.Init.HwFlowCtl = UART_HWCONTROL_NONE;
uartInitStruct.Init.OverSampling = UART_OVERSAMPLING_8;
uartInitStruct.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
uartInitStruct.hdmatx = DmaTx;
uartInitStruct.hdmarx = DmaRx;
uartInitStruct.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_
INIT;

retVal = HAL_UART_Init(&uartInitStruct);
assert_param(retVal == HAL_OK);
```

10. Depending on the desired transmit method (such as polled, interrupt-driven, or DMA), some additional setup will be required; this setting-up is typically performed immediately before beginning a transfer.

The chapter's example-programs initialize USART2 by calling `STM_UartInit`.

UART4 is initialized by calling `SetupUart4ExternalSim` (defined in `Src/Uart4Setup.c`). `SetupUart4ExternalSim` does two things:

- It sets up the MCU's DMA peripheral to generate the data that UART4 will send.
- It calls `STM_UartInit` to initialize UART4.

The UART's setup is not fully explained here. The parts that are not explained can be treated as a *black box*. It's not necessary to fully understand the UART setup in order to work with the example-programs.

In this chapter, we'll develop two types of UART drivers – polled and ISR-based. Next, let's see how all of this plays out by creating a simple polled driver to read data coming into USART2.

Creating a polled UART driver

The chapter's first example-program is in `Src/mainUartPolled.c`, and it's explained in the present section. The program has a task named `polledUartReceive`, which reads bytes from USART2, and puts the bytes in a queue. The task polls the USART2, to read bytes from it. There is another task named `uartPrintOutTask`, and it receives bytes from the queue. The task also displays the bytes in the SystemView app. The task that reads from USART2 serves as a UART **driver**, and we'll refer to it as a **polled UART-driver.**

The chapter's second example-program is presented later. It uses an ISR to read bytes received by USART2, and the ISR puts each byte in a queue. The ISR serves as a UART driver, and we'll refer to it as an **ISR-based UART-driver**. We'll see how using an ISR increases performance over polling. We'll also see how FreeRTOS APIs can be used in ISRs.

Understanding the example-program

When writing low-level drivers, it's a *must* to read through the datasheet in order to understand how the peripheral works. Even if you're not writing a low-level driver from scratch, it is always a good idea to gain some familiarity with the hardware you'll be working with. The more familiarity you have, the easier it will be to diagnose unexpected behavior, as well as to create efficient solutions.0



You can read more about UART peripherals in the STM32F76xxx reference manual, in its chapter on USARTs (chapter 34).

Our first **driver** is shown below. It takes an extremely simple approach to getting data from the UART and into a queue. The queue can be easily monitored and consumed by any task in the system. By monitoring the **receive not empty** (`RXNE`) bit of the UART peripheral's **interrupt status register** (`ISR`), the driver can determine when a new byte is ready to be transferred from the **receive data register** (`RDR`) of the UART into the queue. To make this as easy as possible, the `while` loop is placed in a lower-priority task, `polledUartReceive`. This will let other higher-priority tasks run.

`STM_UartInit` is called to set up and start USART2. This is the UART that receives data. `STM_UartInit` was described earlier.

```
void polledUartReceive( void* NotUsed )
{
    uint8_t nextByte;
    // Setup UART
    STM_UartInit(USART2, 9600, NULL, NULL);
    while(1)
    {
        while(!(USART2->ISR & USART_ISR_RXNE_Msk));
        nextByte = USART2->RDR;
        xQueueSend(usart2_BytesReceived, &nextByte, 0);
    }
}
```

There is another simple task in this example as well; it monitors the queue and prints out whatever has been received.

```
void uartPrintOutTask( void* NotUsed)
{
    char nextByte;
    while(1)
    {
        xQueueReceive(usart2_BytesReceived, &nextByte, portMAX_DELAY);
        SEGGER_SYSVIEW_PrintfHost("%c ", nextByte);
    }
}
```

UART4 is set up and started in `main()`, by calling `SetupUart4ExternalSim`, as shown below. `SetupUart4ExternalSim` was described earlier.

```
SetupUart4ExternalSim(9600);
```

Now that our driver is ready, let's see how it performs.

Running the example-program

The example-program has a third task, named `startUpTask`. The task's purpose is to allow the SystemView app to be started before the other two tasks run. `startUpTask` is the highest-priority task. It spins until SystemView is started in Record mode. The task then deletes itself.

```
void startUpTask( void* NotUsed )
{
    // Indicate startUpTask has started
    BlueLed.On();
    // Spin until the user starts the SystemView app, in Record mode
    while(SEGGER_SYSVIEW_IsStarted()==0){
        lookBusy(iterationsPerMilliSecond);
    }
    BlueLed.Off();
    vTaskDelete(NULL);
}
```

To run the example-program:

1. Download and start the executable using Ozone. The blue LED should turn on immediately.
2. End the Ozone debug-session by clicking on the red on/off icon. This is necessary to prevent SystemView from experiencing data-overflow.
3. Start SystemView in Record mode.

The SystemView messages are shown below. UART4 repeatedly sends the string "`data from uart4`," and the string gets displayed in SystemView, one byte at a time. The string's null-terminator is also sent. This is why there is a gap in the messages, between **uart4** and **data** (the gap is the message with timestamp 0.321 276 773).

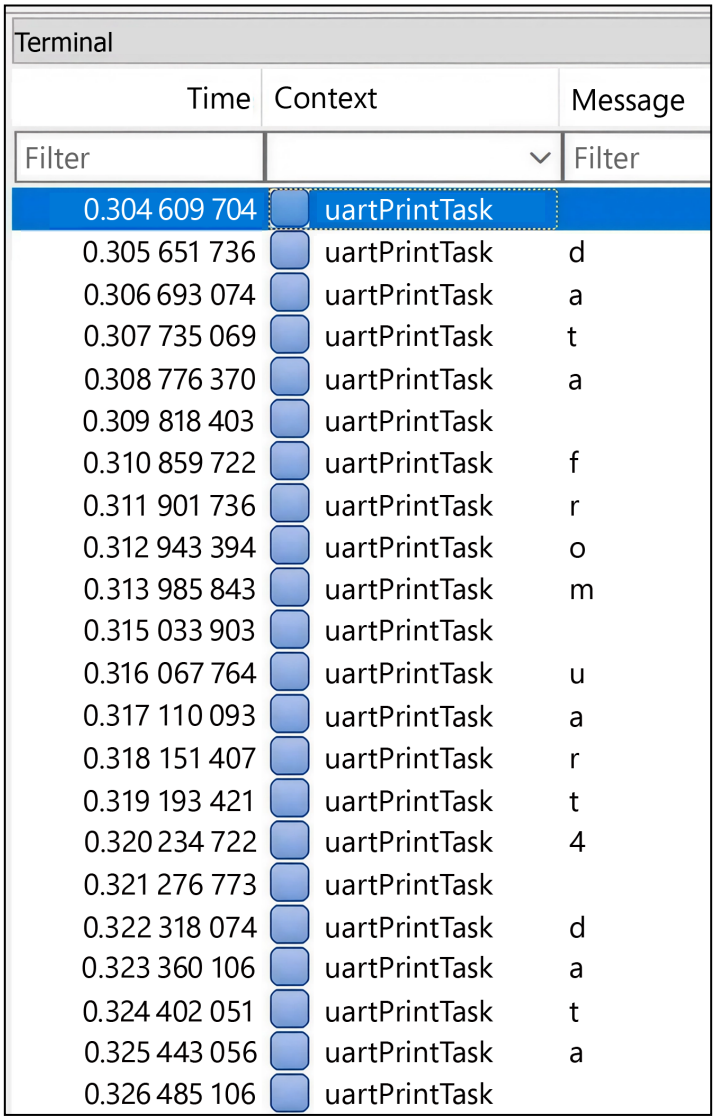


Figure 10.5 : SystemView messages from uartPrintOutTask

Analyzing the performance

The preceding code (`uartPolled`) can be programmed onto the MCU, and we can take a look at the performance using SEGGER SystemView. (*Figure 10.6* is also expanded in *figure 10.6b*.)

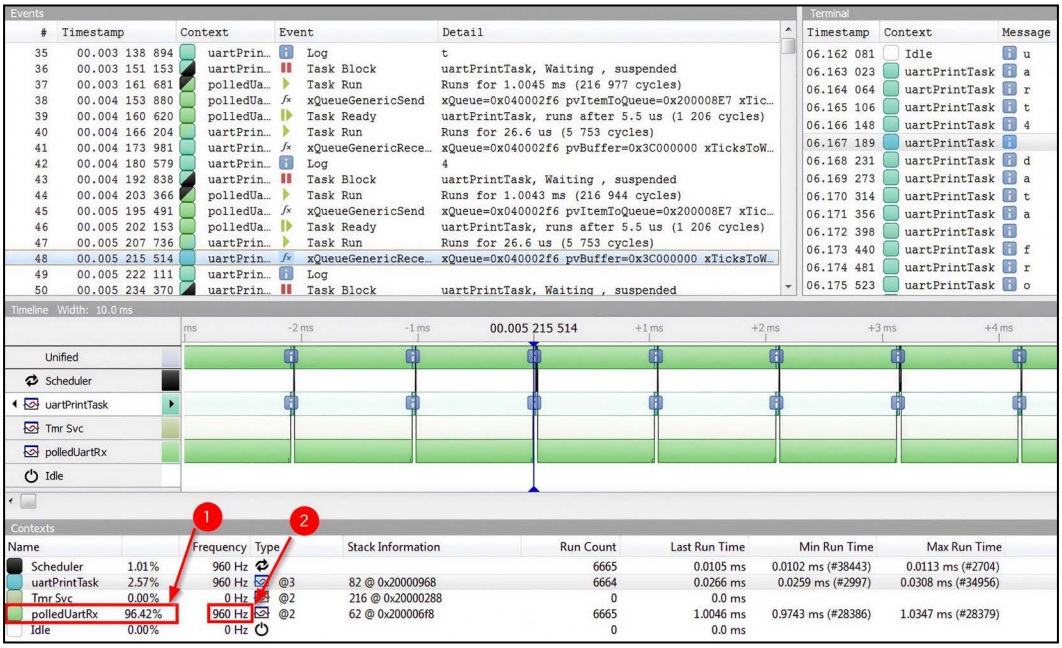


Figure 10.6 : Performance analysis of the polled-based driver

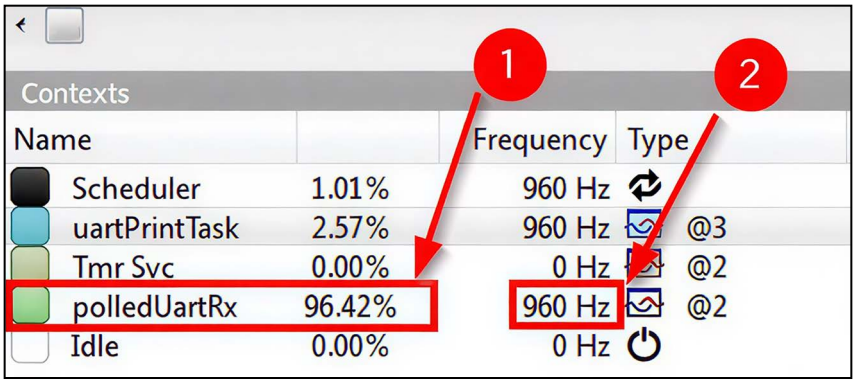


Figure 10.6a : The left corner of figure 10.6

After looking at the execution using SystemView, we quickly realize that – although easy to program – this driver is *horrifically inefficient*. In the figure, the numbered call-outs show:

1. SystemView reports that this driver is utilizing *over 96%* of the CPU's resources. The second column is **CPU Load** (though its header isn't shown here).
2. The queue is being called at 960 Hz (which makes perfect sense given the initial baud rate of 9,600 baud).

You may notice some minor differences between the Context window shown here, and your SystemView. A likely cause is different SystemView versions.

For example, your SystemView display might not have the **Frequency** column that is shown in the figure at the call-out 2. An **Activations** column will probably be displayed instead. In this case, a frequency can be calculated by dividing an **Activations** value by the recording-time's duration. The duration can be found by clicking on **Window** : **System**.

We can see that, while easy to implement, this polling solution comes with significant performance penalties – all while servicing a fairly slow peripheral. Drivers that service peripherals by polling have this trade-off between simplicity and efficiency.

Pros and cons of a polled driver

Here are some of the advantages of using a polled driver:

- It is easy to program.
- Any task has immediate access to data in the queue.

At the same time, there are several issues with this approach:

- It must be one of the highest-priority tasks in the system.
- There is a high chance of data loss when not executing at high priority.
- It is extremely wasteful of CPU cycles.

In this example, we're only transferring data at 9,600 baud. Granted, most of the time was spent spinning on the `RXNE` bit, but transferring every byte as it is received into a queue is also fairly expensive (when compared to pushing bytes into a simple array-based buffer). To put this into perspective, USART2 on STM32F767 running at 216 MHz has a maximum baud rate of 27 Mbaud, which would mean we would need to add each character to the shared queue nearly 3 million times a second (it is currently adding < 1,000 characters per second). Transferring this much data through a queue quickly isn't feasible on this hardware since queue additions take 7 µs each (even if the CPU was doing nothing else, we'd be capable of transferring less than 143,000 characters per second into the queue).

More importantly, there are few opportunities to speed up this polled approach, since we may receive a new character once every millisecond. If any other task was executing for more than 1 ms, the peripheral could potentially be overrun (a new byte is received and overwrites the buffer before the previous byte is read). Because of these limitations, there are very specific circumstances where polled drivers are most useful.

The use of polled drivers

There are a few circumstances where polled drivers are especially helpful:

- **System verification**: It is acceptable to take the performance penalty of a polled driver when performing initial system verification. Though, at such an early stage of development, it is debatable whether an RTOS should be used at all.
- **Simple applications**: If the application happens to be simple, and there is nothing else to be done while waiting for data to be transferred, and there are no power considerations, then it may make sense to use a polled driver.
- **Special cases**: There may be times when a very special-purpose piece of code is needed. For example, a peripheral may need to be serviced with an extremely low amount of latency. In other cases, events could happen extremely quickly. When events are happening in the order of nanoseconds (ns) or microseconds (µs) (instead of ms, as in the previous example), it often makes more sense to simply poll for the event, rather than create a more elaborate synchronization scheme. Since polling may block other code from running, it is important to carefully consider the use of polling and to clearly document it.
- **Transmitting bursts of data**: When transmitting a small burst of data it is often tolerable to use a polling approach. Typically, data is placed into the buffer of a peripheral (e.g., a UART). Then, the status bits of the peripheral are polled until they indicate that the buffer is free for more data. This process continues until all of the data has been placed into the buffer. Polling may continue to be used, waiting for the buffer of data to be fully transmitted by the peripheral.

The bottom line is that, while simple, polling consumes CPU resources and must be used carefully. It is often preferable to use an alternative approach that does not waste resources by polling. To take advantage of any of the non-polled approaches, we'll need to develop another skill – programming ISRs.

FreeRTOS-tasks vs. ISRs

This section compares FreeRTOS-tasks and ISRs. In addition, there are FreeRTOS APIs that can be called from ISRs. An overview of those APIs is presented. The relationship between STM32F767 interrupt priorities and FreeRTOS API calls is also presented.

Differentiating between tasks and ISRs

Before we jump into coding a peripheral driver that uses interrupts and ISRs, let's take a quick look at how interrupts compare to FreeRTOS tasks.

There are many similarities between tasks and ISRs:

- Both provide a way of achieving **parallel** code execution.
- Both only run when required.
- Both may take priority over other tasks and therefore require synchronization methods when sharing data or peripherals with other tasks.
- Both can be written with C/C++



Years ago, when MCUs had limited resources, it was typical for ISRs to be written in assembly language to maximize code efficiency. Also, some of the entry/exit procedures required the execution of specific assembly instructions, e.g., RTI (Return From Interrupt). Modern MCUs have more resources (especially bandwidth), which makes hand-crafting ISRs in assembly language less of a necessity (though sometimes it still makes sense). Also, MCUs like the Cortex-M added specific features to the core to minimize the need for special assembly instructions.

But there are also many differences between tasks and ISRs:

- **ISRs are brought into context by hardware; tasks gain context by the RTOS kernel**: Tasks are always brought into context by the FreeRTOS kernel. Interrupts, on the other hand, are generated by hardware in the MCU, which causes the core to bring an ISR into context.



For an ISR to come into context, its corresponding interrupt must be enabled. The interrupt also needs to be the highest priority of all of the enabled interrupts. Typically, there are registers in the MCU that allow for a) the configuration of the interrupts' priorities, and b) the enabling or disabling of specific interrupts.

- **ISRs typically must be short in duration; tasks, on the other hand, can use whatever time is intended for them**: FreeRTOS tasks are often set up to run in a similar way to an infinite `while` loop – they will be synchronized with the system using primitives (such as queues and semaphores) and switched into context according to their priority. Because tasks can be interrupted by higher-priority tasks and by ISRs, it is acceptable if tasks take many MCU cycles to perform their duties. ISRs, on the other hand, interrupt the presently running task (assuming the kernel is configured to run at the lowest interrupt-priority) and maintain ownership of the MCU. This blocks both tasks and lower-priority ISRs from running while the ISR is executing. Because of this, it is generally necessary to keep the ISRs as short as possible.
- **ISR functions do not take input parameters; tasks can**: Unlike tasks, an ISR's function does not have input parameters passed to it when it is invoked. Since an interrupt is triggered because of a hardware state, the most important job of the ISR is to read the hardware state (through memory-mapped registers) and take the appropriate action(s). For example, an interrupt can be generated when a UART receives a byte of data. In this case, the ISR would read a status register, read (and store) the byte received in a static variable, and clear the interrupt.



Most (but not all) peripherals on STM32 hardware will automatically clear interrupt flags when certain registers are read. Regardless of how the interrupt is cleared, it is important to ensure the interrupt is no longer pending – otherwise, the interrupt will fire continuously and you will always be executing the associated ISR!

- **ISRs may only access a limited ISR-specific subset of the FreeRTOS APIs**: Many of the FreeRTOS APIs may not be called from an ISR. ISRs typically operate at a priority above the tasks, and they use a separate stack. Therefore, ISRs are unable to interact with the FreeRTOS kernel in the same way as tasks. There is a subset of FreeRTOS APIs that are available for use in ISRs, and these APIs have a limited subset of functionality (for example, they are not able to block). An example is `xQueueSendFromISR`, which is like `xQueueSend`. The former can be called from an ISR, but the latter cannot. FreeRTOS has a level of safety built-in, so that calling a non-ISR FreeRTOS API from inside an ISR will cause FreeRTOS to trigger `configASSERT`.



It is important to understand that ISR priorities are independent of RTOS priorities. The ISR priorities determine which interrupts, and therefore ISRs, have higher priority relative to each other. The RTOS priorities, on the other hand, determine which tasks have higher priority relative to each other. FreeRTOSConfig.h uses `configMAX_PRIORITIES` to configure the number of task priorities available to the application.

- **ISRs may operate completely independently of all RTOS code**: There are many cases where an ISR doesn't *need* to interact with the FreeRTOS API at all. In this case, the ISR simply executes as it normally would without an RTOS present, as if it was *bare-metal*. The kernel never gets involved, which makes it very convenient for creating flexible solutions that blend high-performing ISRs (operating completely *above* the RTOS) with tasks.



It is typical for an RTOS to disable interrupts during critical sections. This adds jitter and/or delay to ISRs. When an ISR has timing constraints that cannot handle extra jitter or delay, it is important to minimize the amount of time that interrupts are disabled. MCUs without an interrupt priority scheme typically disable global interrupts during critical sections, which impacts *all* maskable interrupts. However, the Cortex-M7 has a priority-level architecture that allows for interrupts at specific priority levels to be disabled rather than the necessity of disabling global interrupts. This implies that with the Cortex-M7, it is possible to implement ISRs that are not impacted by the critical sections of the RTOS.

- **All ISRs share the same system stack; each task has a dedicated stack**: Each task receives a private stack, but all of the ISRs share the same system stack. When writing ISRs, you'll need to ensure that you reserve enough stack space to allow the ISRs to run, possibly simultaneously, if they are nested.

Now that we've covered the differences between tasks and ISRs, we need to explain a little bit about the interaction between FreeRTOS and ISRs.

Calling FreeRTOS APIs from ISRs

Most of the FreeRTOS APIs covered so far have ISR-safe versions. For example, `xQueueSend()` has an equivalent ISR-safe version, `xQueueSendFromISR()`. There are a few differences between the ISR-safe version and the standard version:

- The `FromISR` variants won't block. For example, if `xQueueSendFromISR` encounters a full queue, it will immediately return.
- The `FromISR` variants require an extra parameter, `BaseType_t *pxHigherPriorityTaskWoken`. It will indicate whether or not a higher-priority task needs to be switched into context immediately following the interrupt.
- The only interrupts that can call a FreeRTOS API are those with a priority that is *logically lower* than `configMAX_API_CALL_INTERRUPT_PRIORITY`. That configuration value is defined in `FreeRTOSConfig.h`. We will explain this more in the next section.

Configuring Interrupts for use with FreeRTOS

The interrupt features of the Cortex-M are very flexible, allowing for different configurations for various use cases. However, with this flexibility comes complexity. The following is an overview of how interrupts are configured in `FreeRTOSConfig.h`, and in the book's example-programs (referred to below as `main_xxx.c`).

Generally, the example-programs use CMSIS library-functions to interact with the **Nested Interrupt Vector Controller (NVIC)** registers. These functions are documented in the CMSIS reference manual. They begin with "`NVIC_`", followed by the rest of the function name. These functions help abstract away some of the complexities inside of the NVIC registers. Some of the important functions are:

- `NVIC_EnableIRQ(interrupt_number)` enables an interrupt in the NVIC. However, there are often other enable-bits in the peripheral that also need to be enabled before the interrupt will initiate an ISR.
- `NVIC_DisableIRQ(interrupt_number)` disables the interrupt in the NVIC.
- `NVIC_SetPriority(interrupt_number, priority)` sets the priority-level of the interrupt relative to other interrupts.



It is *very* important to ensure that an interrupt's priority is properly configured *before* enabling the interrupt.

The Cortex-M allows for the assigning of a priority-level to each interrupt in the NVIC. Each interrupt has a corresponding **Interrupt Priority Register** in the NVIC. ARM defines 8 bits of this register as the priority level. The lowest priority level is 255, and the highest priority level is 0. Each MCU vendor has the freedom to implement less than the 8-bits defined by ARM. For example, our device has only 4 bits implemented, for a total of 16 priorities. The code on our device uses 15 for the lowest priority and 0 for the highest priority.



The fact that, on the Cortex-M, the priority number decreases as the priority level increases has caused much confusion in the embedded programming world.

There have been several good articles written to try and clear the confusion. It is helpful to remember this inverse relationship.

All the interrupts that have a configurable priority-level are maskable. This means that they may be disabled so that the corresponding ISR does not get called when the interrupt occurs. The Cortex-M7 also contains non-maskable interrupts. These interrupts may not be disabled, and their corresponding ISRs will get called when the interrupt occurs (the only exception is if a higher-priority interrupt's ISR is already occurring). The non-maskable interrupts in the Cortex-M7 have *negative* priority levels. These interrupts are usually reserved for use by the Cortex-M7 core.

The Cortex-M7 provides a register that defines which bits of the Interrupt Priority Register are used for the **group-priority field** and which bits are used for the **sub-priority field**. This is accomplished through setting the **PRIGROUP** bits of the **Application Interrupt and Reset Control Register**. This register allows for 8 different ways to partition the priority bits between group-priority and sub-priority. Interrupts having the same group priority may not interrupt each other. The sub-priority field arbitrates between interrupts that have the same group priority. In addition:

- Some ARM documentation refers to the group-priority field as the **pre-emption field**.
- Our code does not make use of sub-priorities, so we configure the `PRIGROUP` bits to 0. This is performed in `main_XXX.c` when a call is made to `NVIC_SetPriorityGrouping(0)`, after the STM HAL initialization is performed. With a setting of 0, the 8 bits of the Interrupt Priority Registers are interpreted as 7 bits of group-priority and 1 bit of sub-priority. This would give, in theory, 128 group-priority levels, with 2 sub-priority levels. However, since our device only implements 4 bits of the Interrupt Priority Register, the effective result is that we have 16 group-priorities and no sub-priorities. We simplify this by saying that we have 16 priority levels. We will use this terminology from here on.

- Since we are not using sub-priority levels, you may wonder which interrupt will have priority when there are two interrupts with the same priority level. In this case, the interrupt with the lower exception-number will take priority over the interrupt with the higher exception-number. However, interrupts with the same priority will not pre-empt each other.



If you are interested in understanding more about group priorities and sub-priorities, see the *Arm v7-M Architecture Reference Manual*, in the section "Exception priorities and preemption".

There is a set of FreeRTOS APIs that can be used in ISRs. FreeRTOS can be configured so that those APIs can only be used with particular interrupt priority-levels, rather than all priority-levels. The priority-levels are specified in `FreeRTOSConfig.h`. (See `Inc/FreeRTOSConfig.h`.) This allows for interrupts higher than a particular priority-level to operate independently of FreeRTOS. The following configuration parameters specify how FreeRTOS interacts with interrupt priorities:

- `configKERNEL_INTERRUPT_PRIORITY` defines the priority-level of the FreeRTOS kernel ISR. Any interrupt at or below the priority-level of the kernel ISR may be disabled temporarily by the FreeRTOS API during its critical sections.
- `configMAX_SYSCALL_INTERRUPT_PRIORITY` defines the priority-level of interrupts whose ISRs can call FreeRTOS API functions. Any ISR whose priority is at or below this level may call FreeRTOS API functions. Also, interrupts at or below this level may be masked by FreeRTOS during its critical sections. Interrupts above this level will not be masked by FreeRTOS during its critical sections, and their ISRs may not call FreeRTOS API functions.

  Some FreeRTOS ports do not implement `configMAX_SYSCALL_INTERRUPT_PRIORITY`, because the MCU does not have the necessary capabilities.

- For the parameters `configKERNEL_INTERRUPT_PRIORITY` and `configMAX_SYSCALL_INTERRUPT_PRIORITY`, you may notice the priority is shifted to the left by `(8-configPRIO_BITS)` bits, resulting in a 4-bit shift. This is because the 4 priority-bits implemented by our device use the upper 4-bits of the Interrupt Priority Register (see *Figure 10.7*).
- `configLIBRARY_LOWEST_INTERRUPT_PRIORITY` and `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY` are not FreeRTOS configuration parameters. They specify priority-levels in non-shifted form and are easier to read. They are used in specifying configuration parameters, such as `configKERNEL_INTERRUPT_PRIORITY`. (See `Inc/FreeRTOSConfig.h`.)



It is *essential* that ISRs do not call FreeRTOS APIs if their interrupt's priority is higher than configMAX_SYSCALL_INTERRUPT_PRIORITY.

The following diagram illustrates the relationship between interrupt priorities and FreeRTOS API calls, for our MCU.

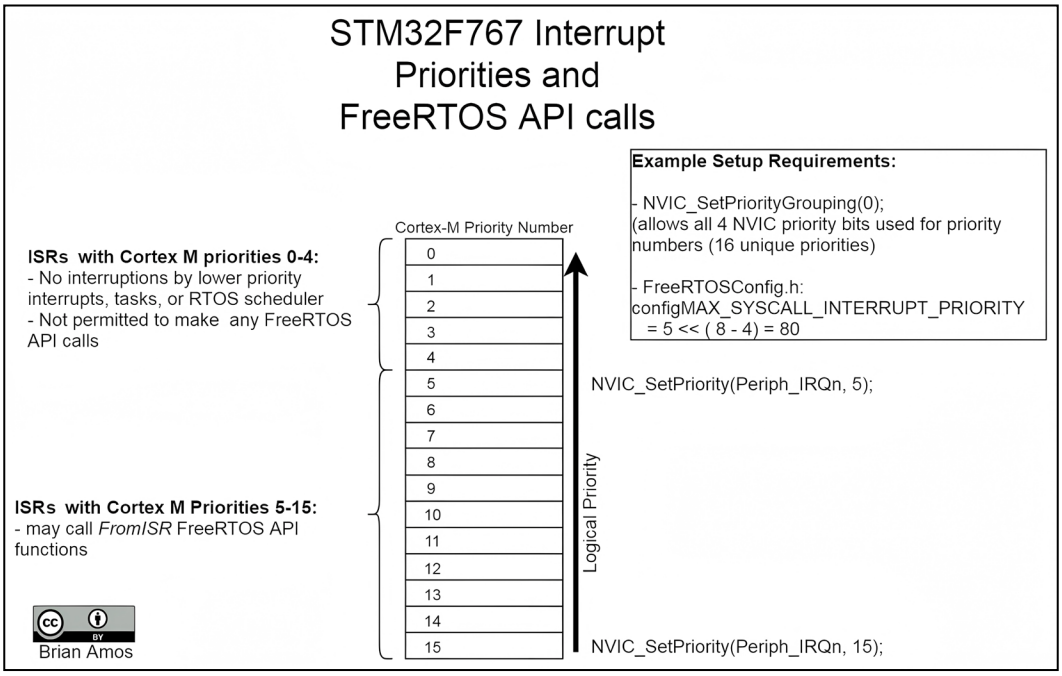


Figure 10.7 : STM32F767 Interrupt Priorities and FreeRTOS API Calls

As we can see in the preceding diagram, there are cases where ISRs can be configured to execute at a priority above anything the RTOS is doing. In our example, ISRs configured with an NVIC priority of 0-4 are identical to traditional "bare-metal" ISRs.



If an ISR calls a FreeRTOS API, and the interrupt has a priority that is logically higher than `configMAX_SYSCALL_INTERRUPT_PRIORITY`, you'll be greeted with a `configASSERT` failure (more details are in *Chapter 16*, *Troubleshooting Tips and Next Steps*.

We now have an understanding of the differences between tasks and ISRs, as well as some of the ground rules for using FreeRTOS API functions from within ISRs. Next, let's take another look at the polled UART-driver, and see how it can be implemented more efficiently as an ISR-based UART-driver.

Creating ISR-based drivers

In the first iteration of the UART driver, a task polled the UART peripheral registers to determine when a new byte had been received. The constant polling is what caused the task to consume more than 95% of CPU cycles. The most meaningful work done by this polled driver was transferring bytes of data out of the UART peripheral and into the queue.

In this iteration of the driver, we'll use an ISR to get data from the UART (`USART2`). Two different ISR-based drivers are implemented, and each uses a different type of data-structure to provide the received data. One implementation uses a FreeRTOS queue, and the other uses a buffer and semaphores.

A queue-based driver

The example-program presented here is in `Src/mainUartInterruptQueue.c`. It has an ISR that receives bytes from `USART2`. The ISR puts the bytes in a FreeRTOS queue. A task removes bytes from the queue and prints them via SystemView.

The example-program is described below. We'll use the same UARTs as in the prior example. The UARTs are configured in the same way, except that `USART2` will generate an interrupt when a byte arrives. The ISR for `USART2` is presented, including how it is configured, how it is built, and how it works.

This example has four primary components:

- `uartPrintOutTask`: This function is the system's task. It does the following: a) initializes `USART2` and associated hardware, b) starts reception in `USART2`, and c) prints anything placed in the `usart2_BytesReceived` queue.
- `startReceiveInt`: This function is called by the task `uartPrintOutTask`. The function sets-up interrupt-based reception for `USART2`.
- `USART2_IRQHandler`: This function is the ISR. It is called when an interrupt occurs for `USART2`.
- `startUart4Traffic`: The function sets-up and starts UART4, and UART4 will continuously send data. This function is called when the timer `oneShotHandle` expires, which is 5 seconds after the FreeRTOS scheduler starts.

The UARTs' baud rate is defined by a macro: `#define BAUDRATE 9600`.

Let's take a look at each component in detail.

uartPrintOutTask

The only task in this example is `uartPrintOutTask`:

```
void uartPrintOutTask( void* NotUsed)
{
    char nextByte;
    STM_UartInit(USART2, BAUDRATE, NULL, NULL);
    startReceiveInt();

    while(1)
    {
        xQueueReceive(usart2_BytesReceived, &nextByte, portMAX_DELAY);
        SEGGER_SYSVIEW_PrintfHost("%c ", nextByte);
    }
}
```

`uartPrintOutTask` does the following:

- Performs the hardware initialization for USART2 by calling `STM_UartInit`.
- Starts an interrupt-based reception for USART2 by calling `startReceiveInt`.
- *Consumes* and prints each character as it is added to the queue `usart2_BytesReceived`, by calling `xQueueReceive`.

startReceiveInt

The `startReceiveInt` function starts interrupt-driven reception:

```
// Global declaration
static bool rxInProgress = false;

void startReceiveInt( void )
{
    rxInProgress = true;

    USART2->CR3 |= USART_CR3_EIE; // Enable error interrupts
    // Enable peripheral and Rx-not-empty interrupts
    USART2->CR1 |= (USART_CR1_UE | USART_CR1_RXNEIE);
```

```
    NVIC_SetPriority(USART2_IRQn, 6);
    NVIC_EnableIRQ(USART2_IRQn);
}
```

`startReceiveInt` sets up everything required to receive data on USART2:

- `rxInProgress` is a flag used by the ISR to determine if the task's queue-receive process has been started. The ISR (`USART2_IRQHandler`) will not attempt to write to the queue until `rxInProgress` is true.
- `USART2` is configured to generate `receive` and `error` interrupts and is then enabled.
- **NVIC_SetPriority** is used to set the interrupt priority. It is a CMSIS API. Since the ISR will call a FreeRTOS API function, the priority must be set *at* or *below* the logical priority defined by `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`. (In `FreeRTOSConfig.h`, it is defined as 5). On Cortex-M processors, smaller numbers signify a higher logical priority. So, assigning a priority of 6 to `USART2_IRQn` will allow the ISR to make calls to FreeRTOS's ISR-safe APIs. (In this case, `xQueueSendFromISR`.)
- Finally, the interrupt-requests generated by USART2 will be enabled by making a call to **NVIC_EnableIRQ**. If `NVIC_EnableIRQ` is not called, USART2 will still generate requests, but the interrupt controller (the "IC" in NVIC) will not *vector* the program-counter to the ISR (`USART2_IRQHandler` will never be called).



In this example, as with nearly all the code in this chapter, we're writing directly to the hardware peripheral registers and not using considerable amounts of abstraction. This is done to keep the focus on how the RTOS interacts with the MCU. If code-re-use is one of your goals, you'll need to provide some level of abstraction above raw registers (or STM HAL code, if you're using it).

USART2_IRQHandler

Here is the code for the ISR, `USART2_IRQHandler`:

```
void USART2_IRQHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    SEGGER_SYSVIEW_RecordEnterISR();

    // The code for error-flag clearing is omitted for brevity
```

```
    . . .

    if( USART2->ISR & USART_ISR_RXNE_Msk)
    {
        uint8_t tempVal = (uint8_t) USART2->RDR;

        if(rxInProgress)
        {
            xQueueSendFromISR(usart2_BytesReceived, &tempVal,
                              &xHigherPriorityTaskWoken);
        }
    }
    SEGGER_SYSVIEW_RecordExitISR();
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Let's take a closer look at each component of the ISR:

- The USART2 registers are directly read to determine whether or not the receive-not-empty flag (`RXNE`) is set. If it is, the contents of the receive data register (`RDR`) are stored to a temporary variable (`tempVal`)—this read clears the interrupt flag. If the receive-process is in progress, `tempVal` is sent to the queue.
- Calls to `SEGGER_SYSVIEW_RecordEnterISR` and `SEGGER_SYSVIEW_RecordExitISR` are made upon entry and exit, which gives SEGGER SystemView the visibility to display the interrupt with all of the other tasks in the system.
- `xQueueSendFromISR` sends the byte read to the queue.

For the API **xQueueSendFromISR()**, the third parameter's name is *pxHigherPriorityTaskWoken. The parameter's use in the ISR is complex, and it is explained below.

In the FreeRTOS reference manual, the parameter's description is, "`xQueueSendFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xQueueSendFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited."

A context switch can be requested by calling the FreeRTOS API `portYIELD_FROM_ISR()`, and passing it `pdTRUE`. In the ISR, the API is called just before exiting, and it's passed the variable `xHigherPriorityTaskWoken`.

```
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
```

When `portYIELD_FROM_ISR()` is called, if its argument is `pdTrue`, it will call the `FreeRTOS` kernel's scheduler to request the context switch.

In the ISR, when xQueueSendFromISR is called, `xHigherPriorityTaskWoken` is passed by reference, for the third parameter (for `*pxHigherPriorityTaskWoken`). (`xHigherPriorityTaskWoken` is initialized to `false` on entry to the ISR.)

In the example-program, there are two tasks: `uartPrintOutTask` and the Idle task. If `uartPrintOutTask` is blocked because it is waiting on the queue, then the Idle task will be the currently-running task. When a byte arrives at USART2, the ISR will run, and it will call `xQueueSendFromISR` to send the byte to the queue. `uartPrintOutTask` will unblock. Since `uartPrintOutTask` has higher priority than the Idle task, `xQueueSendFromISR` will return with `xHigherPriorityTaskWoken` set to `pdTRUE`. When the ISR calls `portYIELD_FROM_ISR`, it will cause the scheduler to make a context switch to `uartPrintOutTask`.

Now that the ISR has been written, we'll need to make sure it will actually be called by the hardware at the appropriate time.

Linking ISRs

This section explains how the linkage is set up for the USART2 ISR, and it provides general tips for **linking ISRs**.

How the ISR is linked

For our ISR to be run, it must have the name `USART2_IRQHandler`.

A system start-up file specifies a default ISR for the system's interrupts, including USART2. The file is `startup/startup_stm32f767xx.s`. If an interrupt is enabled, and an ISR is not provided for the interrupt, then the default ISR will be used for the interrupt. The default ISR is shown below. It is simply an infinite loop, which can be used for debugging purposes.

```
    .section  .text.Default_Handler,"ax",%progbits
Default_Handler:
Infinite_Loop:
  b  Infinite_Loop
  .size  Default_Handler, .-Default_Handler
```

The start-up file also specifies the linker commands for the system's ISRs. The linker commands are shown below for USART2. The linkage is specified as `.weak`. To override the weak linkage, just implement a function with the same name anywhere in the IDE-project. The linker will use that function instead of the weak one.

```
    .weak      USART2_IRQHandler
    .thumb_set USART2_IRQHandler,Default_Handler
```

The linker also puts the ISR's address in the interrupt vector table, in the entry for the interrupt. The interrupt vector table is specified in the start-up file. The entry for the USART2 interrupt is:

```
  .word     USART2_IRQHandler     /* USART2                        */
```

Tips for linking ISRs

When writing ISRs from scratch (as we've done in this example), one area that can prove to be a source of unexpected trouble is ensuring your ISR is properly linked in (and executed). That is, even if you've properly set up the peripheral to generate interrupts, your new ISR might never be called because it isn't named properly. (Instead, the default implementation, defined in a startup file, will likely be called.) Here are some tips to make sure that your shiny new ISR can be found and properly linked in with the rest of the application:

- STM32 `*_IRQHandler` function-names *usually* contain the *exact* peripheral name from the datasheet as a sub-string. For example, USART2 maps to `USART2_IRQHandler` (notice the "S"), and UART4 maps to `UART4_IRQHandler` (no "S" in the peripheral or function name).
- When writing a new implementation for an ISR, it is a good idea to copy and paste the exact `*_IQRHandler` name from the startup file. This reduces the chance of typos, which can cause debug headaches!
- STM start-up files implement a default handler for every interrupt, and the default handler is an infinite loop. If you notice your application becoming unresponsive, it is possible you've enabled an interrupt and your `*_IRQHandler` definition isn't being linked in properly.
- If you happen to be implementing the `*_IRQHandler` inside a C++ file, be sure to use `extern "C"` to prevent *name mangling*. For example, the USART2 definition would be written as `extern "C" void USART2_IRQHandler(void)`. This also means the ISR definition must *not* be inside a class.



When implementing ISRs, take your time and be sure to get the details right (such as the *exact* name). Don't rush into attempting to debug the rest of your application without first ensuring the ISR is called when expected. Using breakpoints inside the ISR is an excellent way of doing this.

startUart4Traffic

The final component that needs to be explored in this example is how data will be sent from UART4 to USART2. These examples are meant to simulate external data being received by USART2. To achieve this without additional hardware, we wired together UART4 Tx and USART2 Rx pins earlier in the chapter. The call to `startUart4Traffic()` is a `TimerHandler` prototype. A one-shot timer is started and set to occur 5 seconds after the application starts.

The function that does all of the heavy lifting is `SetupUart4ExternalSim()`. It is defined in `Src\Uart4Setup.c`, and it was described earlier, in the section "Setting up the UART". It sets-up a continuous circular DMA transfer (which executes without CPU intervention), and a string is transmitted repeatedly by UART4. The string is "`data from uart4`", with a null terminator. A full example using DMA will be covered in the next chapter – for now, it is sufficient to realize that data is being sent to USART2 without involvement from the CPU.



`startUart4Traffic()` creates a *continuous* stream of bytes that will be transmitted out of UART4 Tx and into UART2 Rx (with no flow control). Depending on the selected baud rate and the amount of time it takes for the receiving code to execute, we can expect that, eventually, a byte will be missed on the receiving side during some examples. Keep this in mind when running examples on your own. The next chapter has a section, "Choosing a driver model", with more details on selecting the appropriate driver type for your application.

Performance analysis

Now, let's take a look at the performance of this implementation by building the example-program, loading it onto the MCU, and using SystemView to analyze the actual execution. Before starting SystemView's Record mode, stop the Ozone debug session. This is needed to prevent overflow in SystemView. Unlike other example-programs, this example-program does not wait for SystemView to be started in Record mode.

The following figure shows SystemView, after the example-program was started and 22k bytes were received. (*Figure 10.8* is expanded in *figure 10.8a*.)

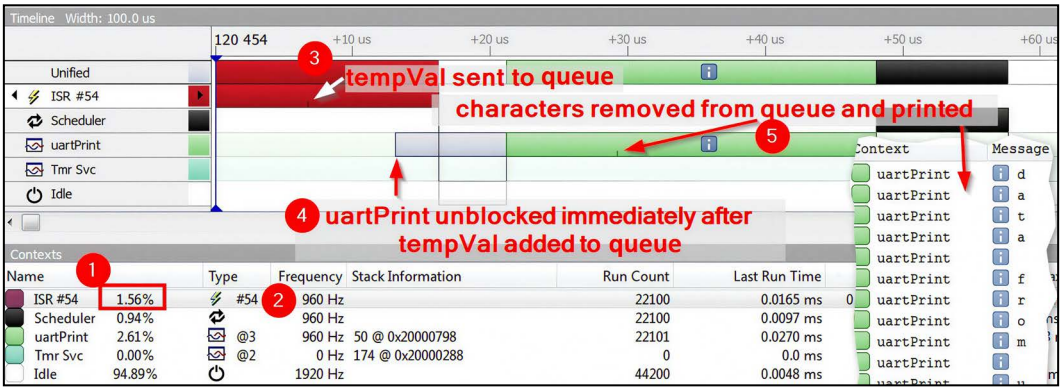


Figure 10.8 : Performance analysis of the queue-based ISR

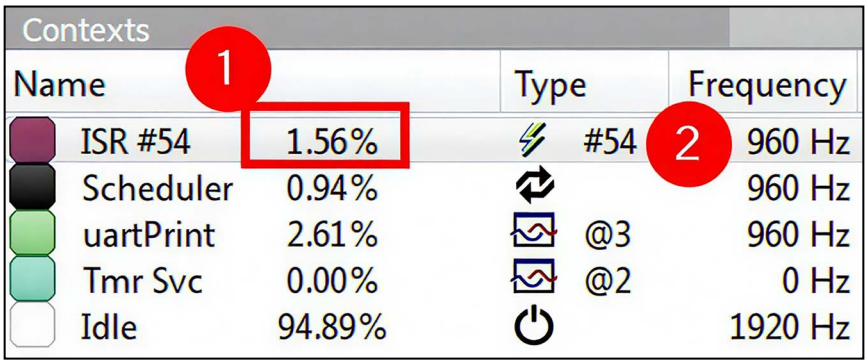


Figure 10.8a : The left corner of figure 10.8

This time around, things look considerably better. Here are some noteworthy items from the preceding screenshot:

1. The ISR responsible for dealing with the incoming data on USART2 Rx is only consuming around 1.6% of the CPU (much better than the 96% we saw when we were using a polled approach). (This is the column **CPU Load**.)
2. The ISR's frequency shows we are still receiving 960 bytes per second – the same as before. (If the **Frequency** column is not displayed, the frequency values can be calculated as described earlier in the chapter.)
3. The small black tick-mark shown here is the exact point in time when `tempVal` is added to `usart2_BytesReceived` by the call to the `xQueueSendFromISR` FreeRTOS API function.

4. We can see the effect of `portYIELD_FROM_ISR` here. The lightly shaded portion of the `uartPrint` task indicates that the task is ready to run. (`uartPrint` is `uartPrintOutTask`.) This happens because `uartPrint` is waiting to receive an item in the queue. The call to `portYIELD_FROM_ISR` forces the scheduler to immediately evaluate which task should be brought into context. The larger portion (starting at ~21 µS) is SystemView's way of signifying that the task is in a *running* state.

   Notice there is a gap between when the ISR exits and `uartPrint` runs. Presumably, the scheduler is being run during this gap, to perform the context switch to `uartPrint`. As we've seen before, SystemView does not always identify when the scheduler is being run.

5. After `uartPrint` begins running, it removes the next character from the queue and prints it using `SEGGER_SYSVIEW_PrintfHost`.

By switching from a poll-based driver to an interrupt-based driver, we've significantly reduced the CPU load. Additionally, systems that use an interrupt-based driver can run other tasks while still receiving data through USART2. This driver also uses a queue-based approach, which provides a very convenient ring buffer, allowing characters to be continuously received and added to the queue, then read whenever it is convenient for higher-level tasks.

Next, we'll work through an example of a similar driver that doesn't use a queue at all.

A buffer-based driver

Sometimes a buffer can be used by a driver instead of a queue. Let's take a look at an example of a **buffer-based driver**. The example-program is identical to the previous one, except that a buffer is used. The example-program is in `Src/mainUartInterruptBuffer.c`.

The buffer is declared as a global byte-array. It is filled by the `ISR USART2_IRQHandler`. When the buffer is full, it is printed by the task `uartPrintOutTask`. After the buffer is printed, it is filled again by the ISR.

```
// The buffer is long enough to hold the string that is
// sent by UART4, plus a null-terminator.
#define BUFFER_LENGTH 17
static volatile uint8_t buffer[BUFFER_LENGTH];
```

Two semaphores are used to coordinate the task and the ISR's access to the buffer. The semaphore `semOkToFill` is taken by the ISR before adding a byte to the buffer. After adding the byte to the buffer, if the buffer is not full, the ISR gives `semOkToFill`. This way, the ISR can add the next received byte to the buffer.

When the ISR fills the buffer, the ISR gives the semaphore `semOkToPrintOut`, so the task can print the buffer. The task takes `semOkToPrintOut` and prints the buffer. The task then gives `semOkToFill`, so the ISR can resume filling the buffer.

uartPrintOutTask

The task `uartPrintOutTask` is shown below. When the task starts, it gives the semaphore `semOkToFill`. This allows the ISR to start filling the buffer. Semaphores are created in the "empty" state, so a semaphore must first be given before it can be taken.

In each iteration of the `while` loop, the task attempts to print the buffer. The task takes the semaphore `semOkToPrintOut`, and waits for the semaphore up to 100 ms. In our testing, the semaphore was always taken in time, after UART4 started sending data. As in the prior example-program, a one-shot timer is created in `main()`. The timer starts UART4's data-transmission, 5 seconds after the scheduler starts.

After taking the semaphore, the task prints the buffer. The task then gives the semaphore `semOkToFill`, so the ISR can start filling the buffer.

```
void uartPrintOutTask( void* NotUsed)
{
    . . .
    // Give the semaphore needed to fill the buffer
    xSemaphoreGive(semOkToFill);
    while(1)
    {
        // Take the semaphore needed to print the buffer
        if( xSemaphoreTake(semOkToPrintOut, 100) == pdPASS )
        {
            semOkToPrintOut_taken++;
            SEGGER_SYSVIEW_Print((char*)buffer);
            // Give the semaphore needed to fill the buffer
            xSemaphoreGive(semOkToFill);
        }
        else
        {
            SEGGER_SYSVIEW_PrintfHost("timeout");
            // * semOkToPrintOut_taken is 0 until the one-shot timer
            //   starts UART4.
```

```
            if (semOkToPrintOut_taken != 0)
                // Record diagnostic data.
                semOkToPrintOut_notTaken++;
        }
    }
}
```

USART2_IRQHandler

Excerpts from the ISR `USART2_IRQHandler` are shown below. After the ISR reads a byte, it takes the semaphore `semOkToFill`, using **xSemaphoreTakeFromISR**. Since `xSemaphoreTakeFromISR` is made for use in an ISR, the API will not block if the semaphore isn't available. In our testing, the semaphore was always available.

The program's algorithm depends on the task being able to print the buffer in the time between when characters arrive. If printing takes longer than that, the semaphore `semOkToFill` will not be available when it is taken, and the received character will not be put in the buffer.

The task uses SystemView to print the buffer. SystemView expects a string with a null terminator, so the ISR formats the buffer accordingly. The string sent by UART4 ends in a null terminator, and the terminator is sent by UART4. When the ISR receives a null-terminator, a "#" character is put in the buffer instead. After the buffer is filled, a null terminator is added to the end. (Null-terminators are processed like this because the buffer can contain the latter part of one message and the beginning of the next message.)

If the received byte fills the buffer, the ISR gives the semaphore `semOkToPrintOut`, using **xSemaphoreGiveFromISR**. The task will then be able to print the buffer. The ISR will not be able to start filling the buffer until after the task prints the buffer and gives the semaphore `rxOkToFill`.

If the received byte does not fill the buffer, the ISR gives the semaphore `semOkToFill`. This allows the ISR to add the next byte to the buffer.

```
void USART2_IRQHandler( void )
{
    . . .
    if( USART2->ISR & USART_ISR_RXNE_Msk)
    {
        // Read the next byte
        uint8_t tempVal = (uint8_t) USART2->RDR;
```

```
        if(rxInProgress)
        {
            // Take the semaphore needed to fill the buffer
            if ( xSemaphoreTakeFromISR(semOkToFill,
                     &xHigherPriorityTaskWoken_take) == pdPASS){
                . . .
                // Replace a null terminator with '#'
                if (tempVal == 0){
                    tempVal = '#';
                }
                buffer[bufferIndex++] = tempVal;
                // Test if the buffer is full
                if(bufferIndex == (BUFFER_LENGTH - 1)){
                    // Add null-terminator to the string
                    buffer[bufferIndex] = 0;
                    bufferIndex = 0;
                    // Give the semaphore needed to print the buffer
                    xSemaphoreGiveFromISR(semOkToPrintOut,
                                          &xHigherPriorityTaskWoken_give);
                }
                else{
                    // Give the semaphore needed to fill the buffer
                    xSemaphoreGiveFromISR(semOkToFill,
                                          &xHigherPriorityTaskWoken_give);
                }
            } // End of: if ( xSemaphoreTakeFromISR. . .
            else {
                // Record diagnostic data
                semOkToFill_notTaken++;
            }
        } // End of: if(rxInProgress)
    } // End of: if( USART2->ISR. . .
    . . .
```

In the ISR, the semaphore APIs have a second parameter, and it's named `*pxHigherPriorityTaskWoken`. The API `xQueueSendFromISR` uses a parameter like that. The parameter was described with the prior example, along with the API `portYIELD_FROM_ISR`.

Just before returning from the ISR, `portYIELD_FROM_ISR` is called, as shown below. It is passed `pdTrue` if any of the semaphore-APIs had returned `pdTrue` via `*pxHigherPriorityTaskWoken`:

```
    . . .
    if ( (xHigherPriorityTaskWoken_give == pdTRUE) ||
         (xHigherPriorityTaskWoken_take == pdTRUE) ){
        xHigherPriorityTaskWoken = pdTRUE;
    }
    else{
        xHigherPriorityTaskWoken = pdFAIL;
    }
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    SEGGER_SYSVIEW_RecordExitISR();
} // End of: USART2_IRQHandler
```

In the ISR, before reading the next byte, the code checks for errors reported by USART2. The code is shown below. (This code was also in the prior example-program, `mainUartInterruptQueue.c`.) Errors are checked by reading the bits overrun (`ORE`), noise error (`NE`), framing error (`FE`), and parity error (`PE`) in the interrupt-state register (`USART2->ISR`). If an error is present, it is cleared by a write to the interrupt-clear register (`USART2->ICR`). Next, the ISR checks whether a new byte has been received (by reading the `RXNE` bit of `USART2->ISR`). If a new byte is available, it is read into `tempVal`.

```
. . .
    // First check for errors
    if( USART2->ISR & ( USART_ISR_ORE_Msk |
                        USART_ISR_NE_Msk |
                        USART_ISR_FE_Msk |
                        USART_ISR_PE_Msk ))
    {
        // Clear error flags
        USART2->ICR |= (USART_ICR_FECF |
                        USART_ICR_PECF |
                        USART_ICR_NCF |
                        USART_ICR_ORECF);
    }

    if( USART2->ISR & USART_ISR_RXNE_Msk)
```

```
    {
        uint8_t tempVal = (uint8_t) USART2->RDR;
        . . .
```

Performance analysis

The example-program can be run in the same way as the prior example-program. As a reminder, before starting SystemView's Record mode, stop the Ozone debug session. This is needed to prevent overflow in SystemView.

Now, let's take a look at the performance of this driver implementation. Intuitively, it might seem that the ISR would be faster when using semaphores and a buffer, compared to using a queue. However, as we'll see, using semaphores and a buffer actually makes the ISR slower. Though, overall, using semaphores and a buffer consumed slightly less of the CPU load than did using a queue.

When using a buffer, the ISR makes two calls to semaphore APIs, whereas when queues are used, only one call is made to a queue API. And, internally, FreeRTOS implements semaphores using queues. This can be seen in the FreeRTOS source-code shown below (from `semphr.h`).

```
#define xSemaphoreTakeFromISR( xSemaphore, . . .)  xQueueReceiveFromISR(. . .)
```

That FreeRTOS implements semaphores as queues can also be seen in the SystemView display shown below. The Event List identifies the API-calls as being calls to queue APIs, even though they are actually calls to semaphore APIs. For example, event **17311** says the call is to `xQueueReceiveFromISR`, but in the source-code, the call is to `xSemaphoreTakeFromISR`.

The SystemView performance data also shows that when using semaphores and a buffer, the ISR is slower. The ISR call at event **17311** took 24 µs, which was typical for ISR calls in our testing. In the prior example, which uses a queue, the ISR-call was typically 16 µs, which is 1.5 times faster.

In the SystemView display, the ISR used 1.84% of the CPU load. In the prior example, the ISR used 1.56%, which is 15% less.

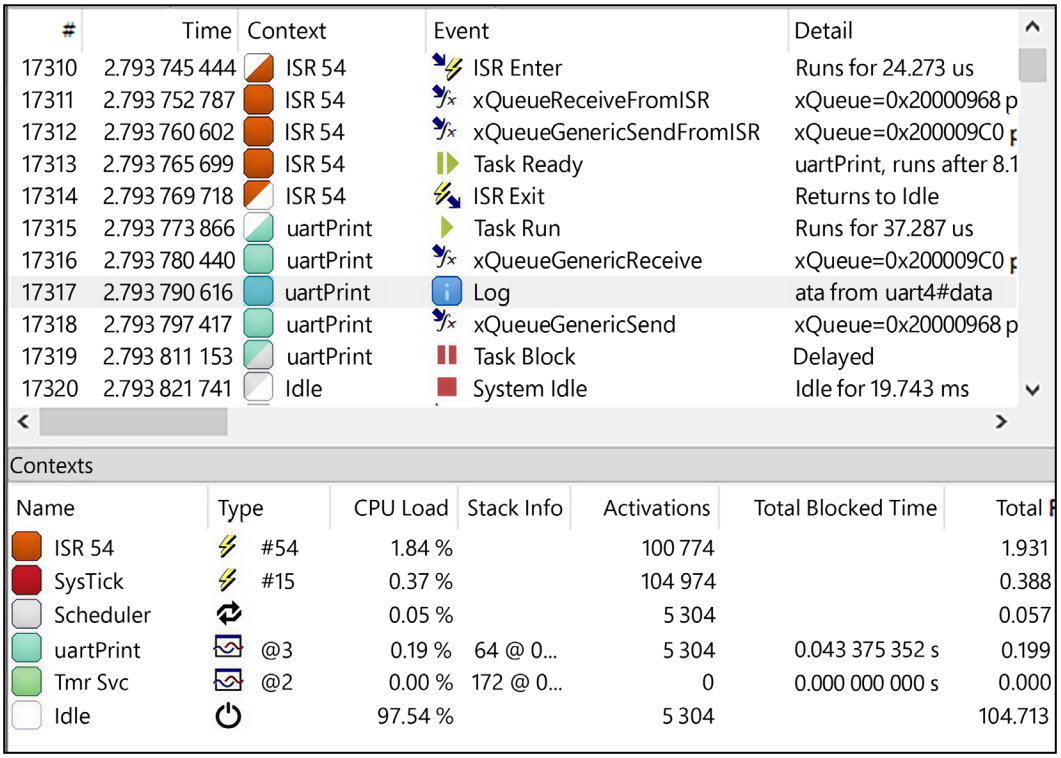


Figure 10.9 : Performance results for using semaphores and a buffer

The biggest performance difference in the use of semaphores and a buffer is that the task printed the whole buffer at a time, and not just one character at a time. Using semaphores and a buffer, the task used just 0.19% of the CPU load. (Shown in the figure above.) Using the queue and printing one character at a time, the task used 2.61% of the CPU load. (Shown in the prior example's performance analysis.) So, printing one character at a time uses 14 times more of the CPU load than printing the buffer.

Overall, using semaphores and a buffer consumed less of the CPU load. This is evident from its Idle task, which used 97.54% of the CPU load. In contrast, using the queue consumed slightly more of the CPU load. There, the Idle task used 94.89% of the CPU load. However, the difference between their Idle-task use is relatively small.

In considering program complexity, the algorithm for using the queue is much simpler than the algorithm for using the semaphores and buffer. In using the semaphores and buffer, it is harder to verify the algorithm works correctly, and thus there is less certainty it is correct.

In choosing between implementing the driver using semaphores and a buffer, or using a queue, the queue would often be the better choice. For the two implementations, their CPU loads are very similar. However, when using a queue, the ISR is 1.5 times faster. Also, the queue implementation is much simpler, and more certain to be correct. (This analysis is specifically for our two example-programs. In general practice, a buffer-based approach is common and a good solution.)

A limitation of our buffer-based driver is that once the buffer fills, it has to be processed before the next byte arrives, or data loss will occur. In practice, such tight timing constraints do not allow high baud-rates, and they are problematic. In the next chapter, we'll see how using two buffers alleviates the problem, and supports much higher baud-rates.

Summary

In this chapter, we introduced two different ways of implementing low-level drivers that interface with hardware peripherals in the MCU. Interrupt-based drivers and polled drivers were covered through examples, and their performance was analyzed and compared using SEGGER SystemView. For interrupt-based drivers, we looked at two ways of implementing them: using semaphores and a buffer, and using a queue.

We discovered that while poll-based drivers are simpler, they consume *a lot* of processor cycles. ISR-based drivers, on the other hand, are a bit more complex, but consume less processor cycles. We also discovered that for our interrupt-based drivers, the implementation that uses a queue would generally be preferable to the implementation that uses semaphores and a buffer.

To get the most out of this chapter, you're encouraged to run through it on actual hardware. The development board was chosen (in part) with the hope that you might have access to Arduino shields. After running through the examples, an excellent next step would be to develop a driver for a shield or another piece of real-world hardware.

In the next chapter, we'll look at additional ways to build a driver for USART2, including the use of DMA and FreeRTOS stream buffers.

Further reading

USARTs:

- The book *Mastering STM32* has a chapter on programming the MCU's UART peripherals, using ST's IDE and HAL.
- ST has a presentation on USARTS: *STM32F7 - USART : Universal Synchronous/Asynchronous Receiver Transmitter Interface*, `https://www.st.com/resource/en/product_training/STM32F7_Peripheral_USART.pdf`

Interrupt priorities:

- The *Armv7-M Architecture Reference Manual* has a section *Exception priorities and preemption* (B1.5.4).
- FreeRTOS.org has a web-page *Running the RTOS on a ARM Cortex-M Core*, and it discusses how the interrupt priorities are configured. `https://www.freertos.org/RTOS-Cortex-M3-M4.html`

Full bibliographic information is in *Appendix B: Reference Information*.

11

More Efficient Drivers and ISRs

In the last chapter, we explored the trade-offs between different drivers for transferring data using a UART. We discovered that we could use ISRs instead of polling to improve the overall efficiency of the driver's usage of processor bandwidth. We also looked at the difference between a queue-based approach and the use of buffers.

In this chapter, we'll attempt to further improve the efficiency of our driver by using **direct memory access (DMA)**. After that, we'll explore how drivers interact with the rest of the system, and we'll take a look at another FreeRTOS feature – **stream buffers**. Throughout this chapter, we'll keep a close eye on overall system-performance, using SystemView. By the end, you should have a good understanding of how to take advantage of RTOS features when writing drivers, and the trade-offs to be made.

The following topics will be covered in this chapter:

- Creating a basic DMA-based driver
- Creating an enhanced DMA-based driver
- Choosing a driver model
- Using third-party libraries and drivers

The chapter's example-programs are in the IDE-project `Chapter_11`. There are two example-programs, and each has its own build-configuration. Two jumper wires are needed for the dev-board, and they were described in the prior chapter.

For the reference manuals cited in this chapter, complete bibliographic information can be found in *Appendix B: Reference Information*.

Creating a basic DMA-based driver

In the prior chapter, we saw that the interrupt-based driver is considerably better than the polled approach, in terms of CPU utilization. But what about applications with a high data-rate that require millions of transfers per second? The next step in improved efficiency can be obtained by having the CPU involved as little as possible, and by pushing the work for transferring data onto specialized hardware in the MCU, namely, the DMA controller.

The example-program presented in this section can be found at `Src/mainUartDMABuff.c`. In presenting it, we'll show how to create a driver that is buffer-based and interrupt-driven. The example-program is the same as the prior example-program, in the previous chapter, except we'll be using the MCU's **DMA** hardware. As in the prior example-program, bytes are received by USART2 and written to a buffer in memory. However, instead of using the CPU to write the received bytes to the buffer, DMA will be used. To do this, we'll need to configure the USART2 and DMA peripherals, accordingly. After the DMA controller is started, it will generate an interrupt once it fills the buffer. The interrupt's ISR will signal a task to print the buffer. The ISR and task use a semaphore to coordinate their shared use of the buffer.

The DMA peripheral's features are extensive and complex. It supports data-transfer between various MCU components, such as memory-to-memory, peripheral-to-memory, and peripheral-to-peripheral. Also, there are many ways the data can be transferred, such as multiple concurrent transfers, FIFO buffers, dual buffers, etc. A comprehensive DMA tutorial could be a small book in itself.

In explaining DMA, we'll focus on the DMA features used in the example-program. The features and their use are explained from a "black box" perspective. The primary settings are described, but not all of the details are explained.

The next section introduces DMA from a conceptual perspective. The following section explains the DMA configuration used in the example-program. Then, the example-program is presented.

Introducing DMA

Remember the assertion that the processor couldn't really do things truly in parallel? This is still true. However . . . modern MCUs contain more than just a processing core. While our processing core is chugging along dealing with instructions, there are many other hardware subsystems hard at work inside the MCU. One of these hard-working subsystems is called a **DMA controller, which is illustrated in the figure below:**

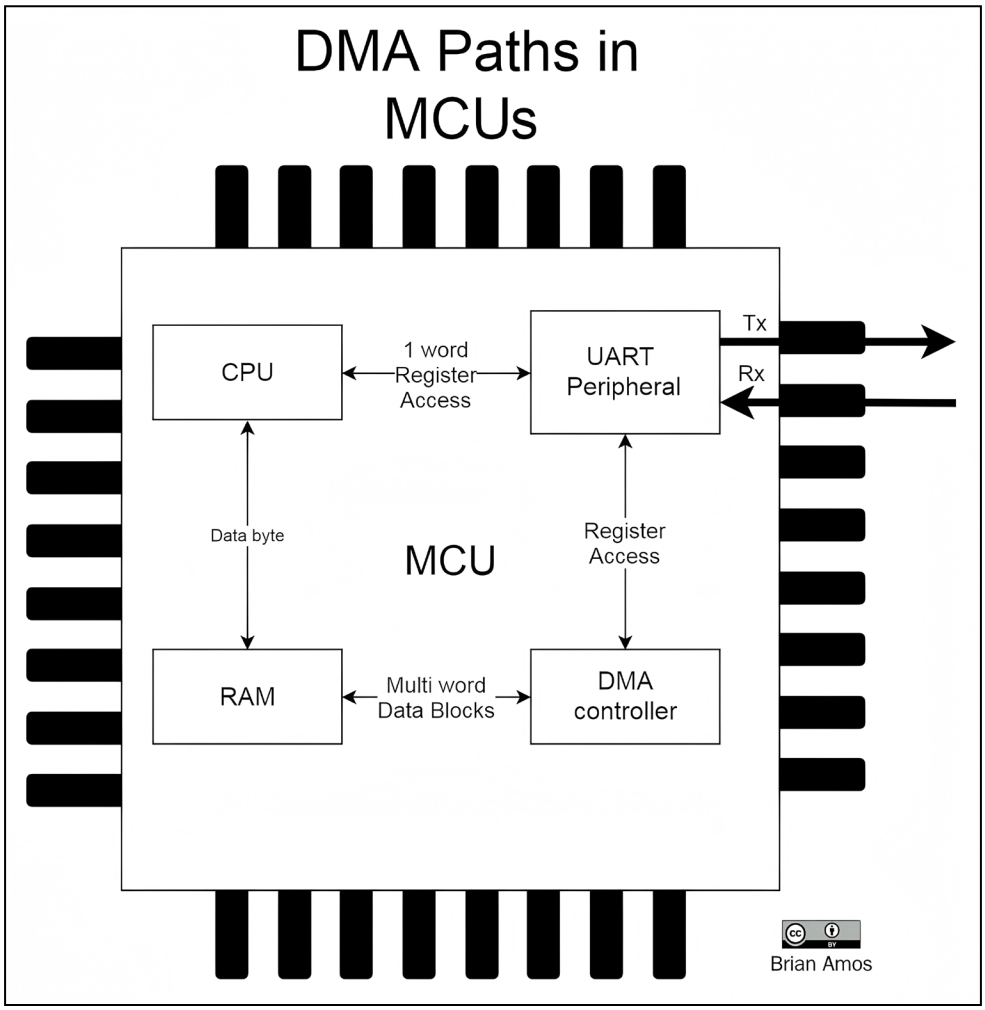


Figure 11.1 : DMA paths in MCUs

The preceding diagram presents a very simplified hardware block-diagram that shows a view of two different data paths available from RAM to a UART peripheral.

In the case of receiving a stream of bytes from a UART without DMA, information from the UART will move into the UART registers, be read by the CPU, and then be pushed out to RAM for storage. In that process:

1. The CPU must detect when an individual byte (or word) has been received, either by polling the UART register flags, or by setting up an ISR that will execute when a byte is ready.
2. After the byte is transferred from the UART, the CPU can then place it into RAM for further processing.
3. Steps 1 and 2 are repeated until the entire message is received.

When DMA is used in the same scenario, the following happens:

1. The CPU configures the DMA controller and peripheral for the transfer.
2. The DMA controller takes care of *all* transfers between the UART peripheral and RAM. This requires no intervention from the CPU.
3. The CPU will be notified when the entire transfer is complete, and it can go directly to processing the transferred bytes.

Most programmers find DMA to be nearly magical if they're accustomed to dealing with super-loops and ISRs. For example, the DMA controller can be configured to transfer a block of memory to the peripheral, as the peripheral needs it, and then provide a notification (typically via an interrupt) when the transfer is complete – that's it!

This convenience does come at a price, of course. It does take some time to set up the DMA transfer initially, so for small transfers, it might actually take more CPU-time to set up the transfer than if an interrupt or polled method was used.

There are also some caveats to be aware of — each MCU has specific limitations. So, be sure to read the details of the datasheet, reference manual, and errata before counting on DMA for a critical system-component:

- The bandwidth of the MCU's internal buses limits the number of bandwidth-hungry peripherals that can be placed on a single bus.
- Occasionally, limited availability of DMA channels to peripherals also complicates the design process.

These types of reasons are why it is important to get all team-members involved with the early-stage design of embedded systems, rather than just throwing it over the wall.

DMA is great for accessing a large number of peripherals efficiently, giving us the ability to add more and more functionality to the system. However, as we start adding more and more modules of code to the embedded system, inter-dependencies between subsystems become more complex as well.

Configuring the DMA peripheral

STM32F767 has two DMA controllers. Each can be used to transfer data within the MCU without involving the processor. We'll use a DMA controller to transfer data from USART2 to memory. Using the MCU's reference manual, we can look up how to use DMA to receive data from USART2, as shown in the following figure. In the figure, reception from USART2 is labeled as `USART2_RX`. The figure shows that DMA1 is used, and the stream and channel used are Stream 5 and Channel 4. (The figure is from ST's MCU reference-manual, *STM32F76xxx and STM32F77xxx advanced Arm-based 32-bit MCUs*, RM0410.)

Table 27. DMA1 request mapping

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	SPDIFRX_DT	SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX	SPDIFRX_CS	SPI3_TX
Channel 1	I2C1_RX	I2C3_RX	TIM7_UP	-	TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1	-	I2C4 _RX	TIM4_CH2	-	I2C4 _RX	TIM4_UP	TIM4_CH3
Channel 3	-	TIM2_UP TIM2_CH3	I2C3_RX	-	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
Channel 5	UART8_TX	UART7_TX	TIM3_CH4 TIM3_UP	UART7_RX	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX	TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	-	TIM5_UP	-
Channel 7	-	TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX
Channel 8	I2C3_TX	I2C4_RX	-	-	I2C2_TX	-	I2C4_TX	-
Channel 9	-	SPI2_RX	-	-	-	-	SPI2_TX	-

Figure 11.2 : DMA stream and channel for USART2_RX

The next figure shows the DMA controller. (This figure is also from ST's MCU reference-manual.) The stream provides a path from the DMA's peripheral port to the memory port. The channel selects the peripheral to be used for the stream.

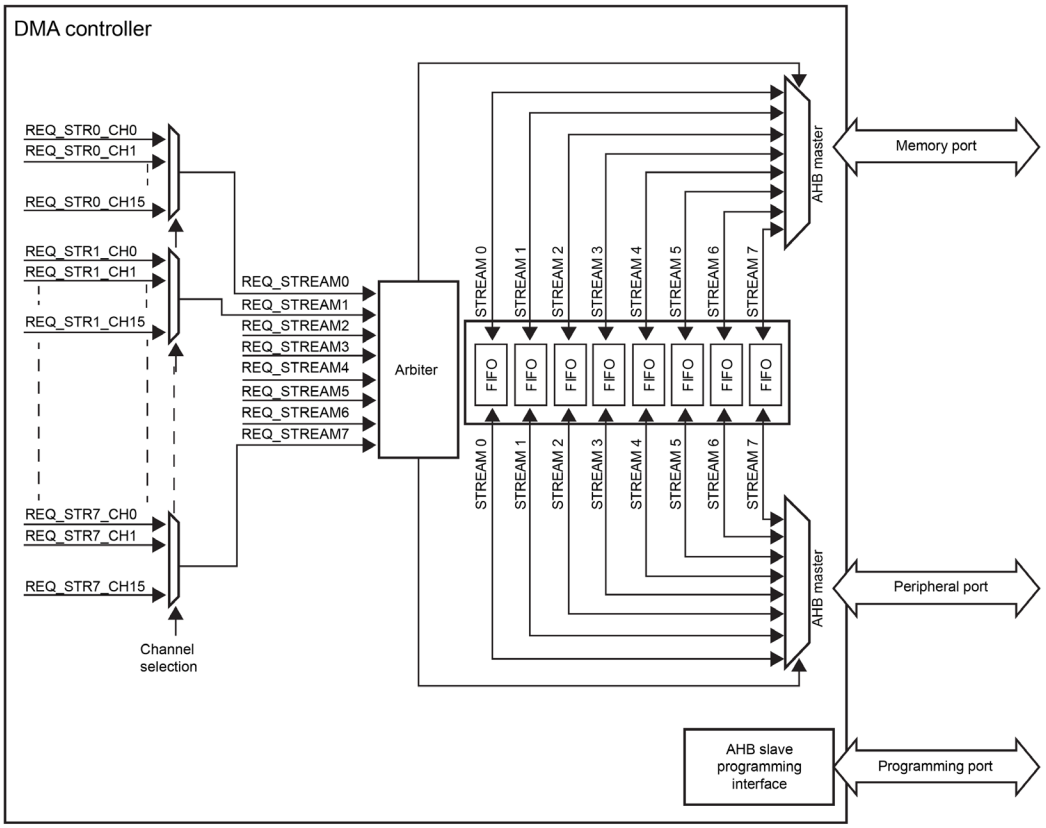


Figure 11.3 : DMA controller, showing streams and channels

Now that we know the channel and stream numbers, we can add some initialization code to set up the DMA1 and USART2 peripherals:

- `DMA1_Stream5` will be used to transfer data from the `receive` data-register of USART2 directly into a buffer in RAM.
- `DMA1_Stream5` will be set up to trigger an interrupt after the entire buffer in RAM has been filled. (Our ISR will process that interrupt.)
- `USART2` will not have interrupts enabled. They are not needed since DMA will perform all transfers from the peripheral register to RAM.

`DMA1` is configured and initialized by the function `setupUSART2DMA()`, in our example-program. The function is shown below, and described.

1. First, the clock to the `DMA1` peripheral is enabled. Then the interrupt-priorities are set up for `DMA1 Stream5`. And, the interrupt for `DMA1 Stream5` is enabled in the NVIC:

```
void setupUSART2DMA( void )
{
    __HAL_RCC_DMA1_CLK_ENABLE();

    NVIC_SetPriority(DMA1_Stream5_IRQn, 6);
    NVIC_EnableIRQ(DMA1_Stream5_IRQn);
```

2. Next, the DMA stream is configured by filling out a `DMA_HandleTypeDef` struct named `usart2DmaRx`. The struct is declared as a global variable, at the top of `mainUartDMABuff.c`:

```
static DMA_HandleTypeDef usart2DmaRx;
```

3. `usart2DmaRx` is filled in, like this

```
    . . .
    // Specify stream 5
    usart2DmaRx.Instance = DMA1_Stream5;

    // Specify channel 4
    usart2DmaRx.Init.Channel = DMA_CHANNEL_4;

    // Specify the data-transfer is from the peripheral to memory
    usart2DmaRx.Init.Direction = DMA_PERIPH_TO_MEMORY;

    // Specify not to use the stream's FIFO queue.
    // * This is referred to as "direct mode".
    // * After each single data-transfer from the peripheral to the
    //   FIFO, the data is immediately stored into the memory
    //   destination.
    usart2DmaRx.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
    // Specify the stream's priority, relative to other streams
    // in the DMA controller.
    usart2DmaRx.Init.Priority = DMA_PRIORITY_HIGH;
    // Specifies the DMA mode.
```

```
    // * Normal mode specifies the data-transfer occurs once
    //   (vs circular mode).
    // * Normal mode also specifies the DMA flow-controller is used
    //   (vs the peripheral flow-controller).
    usart2DmaRx.Init.Mode = DMA_NORMAL;

    //
    // Peripheral-transfer configuration
    //

    // DMA Peripheral data size.
    // * Specifies one byte.
    usart2DmaRx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;

    // Specify that the peripheral address is not incremented
    // after each transfer.
    usart2DmaRx.Init.PeriphInc = DMA_PINC_DISABLE;

    // Specify the DMA controller should generate a "single
transfer".
    // * Each DMA request generates a data transfer of a byte.
    // * Since direct-mode is used, "single transfer" must be used.
    usart2DmaRx.Init.PeriphBurst = DMA_PBURST_SINGLE;

    //
    // Memory-transfer configuration
    //

    // DMA memory data-size
    // * Specifies one byte.
    usart2DmaRx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;

    // Enable memory incrementing.
    // * The DMA controller will increment the memory address
    //   after a transfer
    usart2DmaRx.Init.MemInc = DMA_MINC_ENABLE;

    // Specify the DMA controller should generate a "single transfer".
```

```
// * Each DMA request generates a data transfer of a byte.
// * Since direct-mode is used, "single transfer" must be used.
usart2DmaRx.Init.MemBurst = DMA_MBURST_SINGLE;
```

4. `HAL_DMA_Init()` is called to initialize the DMA stream specified in `usart2DmaRx`:

```
retVal = HAL_DMA_Init(&usart2DmaRx);
assert_param( retVal == HAL_OK );
```

5. The transfer-complete interrupt is enabled for `DMA1_Stream5`. The interrupt's ISR is `DMA1_Stream5_IRQHandler`. The ISR is defined in the program-file (`mainUartDMABuff.c`), and it is presented later.

```
DMA1_Stream5->CR |= DMA_SxCR_TCIE;
```

6. For USART2, DMA-mode is enabled for reception:

```
USART2->CR3 |= USART_CR3_DMAR_Msk;
```

Here's a summary of the most relevant portions of the DMA stream:

- `DMA1_Stream5` is set as the instance. All calls that use the `usart2DmaRx` struct will reference Stream 5.
- Channel 4 is attached to stream 5.
- The peripheral address is not incremented after each transfer – the address of the USART2 receive data register (`USART2->RDR`) doesn't ever change.
- Memory incrementing is enabled for the write to RAM. The DMA hardware will automatically increment the memory address after a byte transfer from USART2, filling the buffer.
- The `transfer-complete` interrupt is enabled for `DMA1_Stream5`.
- USART2 is set-up for `DMA receive-mode`. It is necessary to set this bit in the USART peripheral configuration to signal that the peripheral's receive register will be mapped to the DMA controller.



Additional details about how this struct is used can be found by looking at the `DMA_HandleTypeDef` struct definition in `stm32f7xx_hal_dma.h`, and `HAL_DMA_Init()` in `stm32f7xx_hal_dma.c`. Cross-reference the registers used by the HAL code with the STM32F76xxx RM0410 reference manual, in chapter 8, on DMA. This same technique is often most productive for understanding *exactly* what the `HAL` code is doing with individual function-parameters and struct members.

Starting a DMA receive

In the example-program, the DMA receive process is started by calling the local function `startReceiveDMA()`, which is shown below. The function has two input parameters: a pointer to the buffer for the received data (`Buffer`), and the buffer's length (`Len`). The function initializes the DMA controller by calling the function `setupUSART2DMA()`, which we just looked at.

`startReceiveDMA()` then calls HAL_DMA_Start to enable the DMA stream. The inputs to HAL_DMA_Start include Buffer and Len. Next, USART2 is enabled, to start receiving data.

The DMA data-transfer from USART2 to memory then begins. Once the buffer is filled, the data transfer stops. To receive another buffer of data, `startReceiveDMA()` must be called again.

Additional information on the example-program's DMA operation can be found in the code comments. For general information on using DMA, ST provides a document titled *Using the . . . STM32F7 Series DMA controller, (AN4031)*.

```
int32_t startReceiveDMA( volatile uint8_t * Buffer, uint_fast16_t Len )
{
    . . .
        // Disables the DMA Stream.
        __HAL_DMA_DISABLE(&usart2DmaRx);

        // Wait for the EN bit to be set to 0.
        while((DMA1_Stream5->CR & DMA_SxCR_EN) != 0);

        // Configure and initialize DMA stream
        setupUSART2DMA();

        // Start the DMA Transfer
        if( HAL_DMA_Start(&usart2DmaRx, (uint32_t)&(USART2->RDR),
                          (uint32_t) Buffer, Len) != HAL_OK )
        {
            . . .
            return -1;
         }

         // Clear error flags.
         USART2->ICR |= (USART_ICR_FECF | USART_ICR_PECF |
```

```
            USART_ICR_NCF | USART_ICR_ORECF);

            // Enable the UART
            USART2->CR1 |= (USART_CR1_UE);
        . . .
}
```

The example-program's DMA-use is a prototype. Optimizations should be possible, and features can be added to make the process more robust, such as handling errors detected by USART2.

Now that the DMA process is presented, we can explore driver-implementations using DMA interrupts instead of UART interrupts.

A buffer-based driver with DMA

The current example-program implements a buffer-based driver that uses DMA. As mentioned earlier, the example-program is similar to the prior chapter's buffer-based example-program. There is a buffer that is shared by a task and an ISR. The task prints the buffer. The task waits on a semaphore, which indicates the buffer is full. The task is named `uartPrintOutTask`, and its code is shown below. The ISR gives the semaphore when the buffer is full. The ISR is for the DMA interrupt that is triggered when the buffer is full. The ISR is named `DMA1_Stream5_IRQHandler`, and it is shown later.

The buffer is declared as a global array, `memoryBuffer`, as shown below. UART4 repeatedly sends a 16-byte string. `memoryBuffer` is made large enough to hold that string, plus an extra byte. The extra byte is used to hold a null-terminator, which is needed to print the buffer using `SEGGER_SYSVIEW_Print`.

```
#define STRING_LENGTH 16
#define BUFFER_LENGTH (STRING_LENGTH + 1)
static volatile uint8_t memoryBuffer[BUFFER_LENGTH];
```

For the task's while-loop, each iteration begins by calling `startReceiveDMA()`, which starts (or restarts) the DMA transfer. `memoryBuffer`'s address is passed to `startReceiveDMA()`.

This driver only needs one semaphore to coordinate the use of the buffer by the task and ISR. The reason is that when the ISR gives the semaphore, the buffer has been filled and the DMA controller has stopped filling the buffer. The DMA controller will not start filling the buffer again until after the task prints the buffer, and calls `startReceiveDMA()`.

```
void uartPrintOutTask( void* NotUsed)
{
    . . .
    // Configure and initialize the DMA stream
    setupUSART2DMA();
    // Initialize USART2, pass usart2DmaRx for use in receive
    STM_UartInit(USART2, BAUDRATE, NULL, &usart2DmaRx);

    while(1)
    {
        // Start the DMA transfer for each iteration of the loop.
        // * After the DMA transfer completes, it must be restarted.
        startReceiveDMA(memoryBuffer, (uint_fast16_t)(STRING_LENGTH));

        // Take the semaphore. The timeout is 100 ticks.
        if(xSemaphoreTake(rxDone, 100) == pdPASS)
        {
            // NDTR is the remaining number of data items to be transferred,
            // 0 signals completion
            if(DMA1_Stream5->NDTR == 0)
            {
                // Code not shown for setting null-terminators in the buffer.
                 . . .
                SEGGER_SYSVIEW_Print("received: ");
                SEGGER_SYSVIEW_Print((char*)memoryBuffer);
            }
            . . .
        }
        else
        {
            stopReceiveDMA();
            . . .
            SEGGER_SYSVIEW_PrintfHost("timeout");
        }
    }  // End of: while(1)
}
```

The ISR is shown below. If a reception is in progress (based on the value of `rxInProgress`) and the transmit-complete flag is set, the following takes place:

- The DMA interrupt flag is cleared.
- The `rxDone` semaphore is given.

This is all that is required when using DMA-based transfers, since the DMA controller does all of the bookkeeping associated with the buffer.

```
void DMA1_Stream5_IRQHandler(void)
{
    . . .
    // HISR is the DMA high interrupt status-register.
    // TCIF5 is the transfer-complete interrupt-flag for Stream 5.

    if(rxInProgress && (DMA1->HISR & DMA_HISR_TCIF5))
    {
        . . .
        // HIFCR is the DMA high interrupt-flag clear register
        // * CTCIF5 clears the transfer-complete interrupt-flag, for Stream 5
        DMA1->HIFCR |= DMA_HIFCR_CTCIF5;

        xSemaphoreGiveFromISR(rxDone, &xHigherPriorityTaskWoken);
    }
    . . .
}
```

This driver's functionality is identical to the functionality of the last example-program in the prior chapter. That example-program also used a buffer and was interrupt-driven. The difference is the DMA version of the driver doesn't interrupt the application every time a byte is received. The only interrupt generated is when the whole buffer has been filled.

Performance analysis

Let's take a look at the performance of the DMA-based implementation compared to the buffer-based approach from the prior chapter.

For the DMA-based driver, performance data was collected for 4 minutes of operation.

Contexts

Name	CPU Load	Activations	Min Run Time	Max Run Time
ISR 32	0.09 %	14 449	0.014 148 ms	0.016 185 ms
SysTick	0.31 %	240 815	0.002 972 ms	0.004 403 ms
Scheduler	0.07 %	14 449	0.010 532 ms	0.011 421 ms
uartPrint	0.28 %	14 449	0.045 458 ms	0.048 435 ms
Tmr Svc	0.00 %	0	0.000 000 ms	0.000 000 ms
Idle	99.25 %	14 449	16.538 079 ms	16.544 583 ms

Figure 11.4 : Performance analysis of DMA-based implementation

For both the DMA-based driver and the prior interrupt-driven driver, UART4 ran at 9,600 baud, and the buffer was 16 bytes.

This time around, we can make the following observations about the overall system behavior:

- The DMA ISR is now consuming 0.09% of CPU cycles (vs. 1.84% previously, a 95% reduction).
- The ISR's frequency is 60 Hz (14,449 activations / 240 sec).

The frequency of the ISR has been reduced to only 60 Hz, from 960 Hz. This is because, rather than creating an interrupt for every byte, there is only an interrupt generated at the end of the transfer of 16 bytes (960 / 16 = 60).

Reducing the ISR's frequency also reduces the frequency of context switches for the Idle task. Although it seems trivial with these simple examples, excessive context-switching can become a very real problem in large applications with many tasks and interrupts.

The overall flow is similar to that of the prior interrupt-driven driver, with the only difference being that there is only a single ISR executed when the buffer is filled (instead of one interrupt for each byte transferred). This can be seen in SystemView. (*Figure 11.5* is expanded in *figure 11.5a*.)

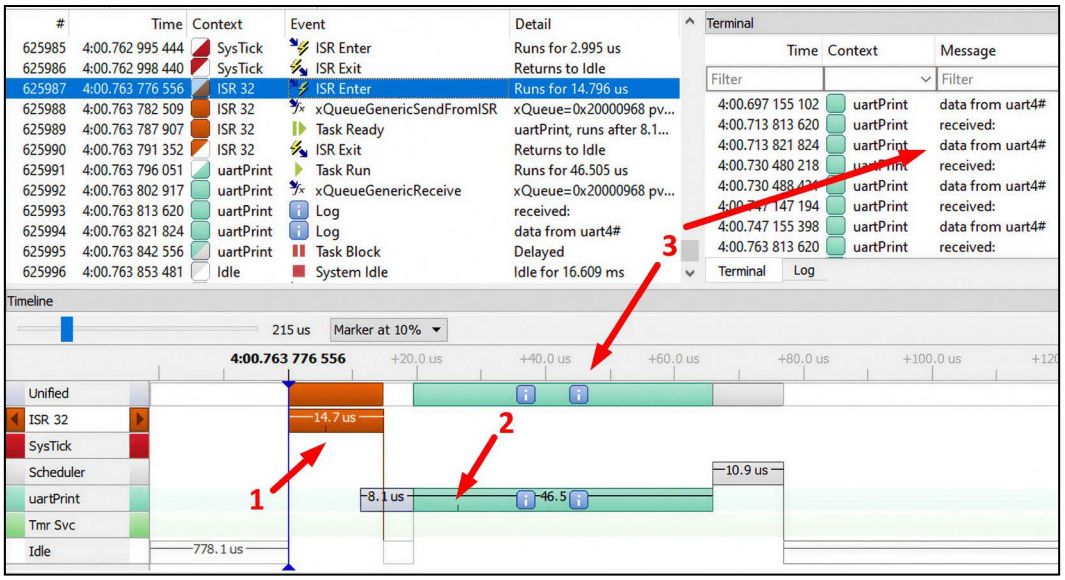


Figure 11.5 : System View display of DMA-based implementation

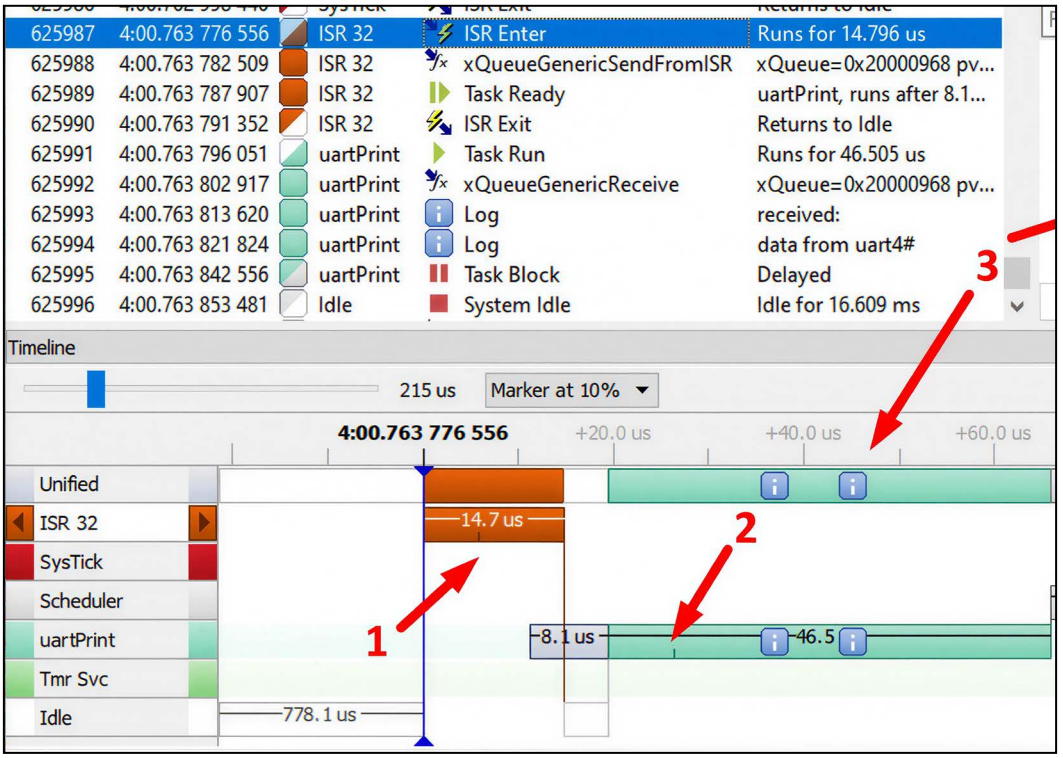


Figure 11.5a : The left side of Figure 11.5

From the preceding screenshot, we can observe the following:

1. The DMA ISR is executed once (after all 16 bytes are transferred into the buffer). A semaphore is shown by the tick mark that arrow 1 is pointing to.
2. The ISR wakes up the blocked `uartPrint` function. Arrow 2 is pointing to the tick mark where the semaphore is taken.
3. The two **i** info-boxes show where the print messages are generated. The remainder of the task's time is spent on restarting the DMA controller for the next transfer, then running `xSemaphoreTake`, which blocks.

The performance data can be used to calculate an upper-bound for the baud-rate that the system can support. Once the buffer is filled, the DMA transfer stops, and the ISR is called. The ISR gives the semaphore, which allows the task to run. The task prints the buffer, and restarts the DMA transfer. So, in the time from when the DMA transfer stops, until when it is restarted, data cannot be transferred from the USART to the DMA stream. We'll refer to this time period as the **transfer-halted time**.

In order to not lose data received by the USART, the transfer-halted time must be less than the arrival time between bytes. This assumes the USART only buffers one received byte, and the DMA controller copies that byte to the stream. It's also assumed that data is arriving continuously, as in our example-program.

The transfer-halted time is approximately the ISR's runtime plus the task's runtime. From our test, the **Contexts** window (*figure 11.4*) shows the ISR runs for at most 0.0162 ms, and the task runs for at most 0.0485 ms. From the **Events** window, it appears the context switch between them is less than 0.005 ms. So, the transfer-halted time is at most 0.0697 ms. 10 bits are sent per byte (including the stop and start bits). This gives an upper-bound on the baud-rate of 143,472 bits per-second:

```
( (1/(0.0697 * 0.001)) [bytes per second] * 10 [bits sent per byte] )
```

A limitation of our buffer-based driver is that once the buffer fills, it has to be processed before the next byte arrives, or data loss will occur. In practice, such tight timing-constraints do not allow high baud-rates, and they are problematic. In the next section, we'll see how using two buffers alleviates the problem, and supports much higher baud-rates.

Even though we've proved to ourselves that it is possible to efficiently transfer data quickly by using DMA, this current setup doesn't have the same convenience that the interrupt-driven queue did. Tasks rely on entire blocks to be transferred, rather than using a queue. This might be fine or might be an inconvenience, depending on what the goals of the higher-level code are.

Character-based protocols will tend to be easier to implement when written on top of a queue-based driver API, rather than a buffer-based driver API (such as the one we've implemented here). However, we saw in the prior chapter that queues become computationally expensive very quickly. Each byte took around 30 µs to be sent to, and removed from, the queue. If transferring data at 128,000 baud, accessing the queue would consume about 3/8 of the CPU load (a new byte is received every 78 µs and it takes 30 µs to process).

In the past, if you really needed to implement a character-oriented driver, you could roll your own highly efficient ring-buffer implementation, and feed it directly from low-level ISRs (bypassing most of the FreeRTOS primitives to save time). However, as of FreeRTOS 10, there is an alternative – stream buffers, which we'll look at next.

Creating an enhanced DMA-based driver

Next, we'll see how our UART-driver's performance can be improved by using FreeRTOS stream-buffers, and also, a continuous DMA transfer that uses two buffers. Stream-buffers are a type of queue. Internally, they are implemented by using task notifications, which makes them relatively fast.

First, stream buffers are presented, and then the DMA enhancements that we'll use. An example-program implements the driver, and the example-program is described here. It is in `Src/mainUartDMAStreamBufferCont.c`.

We'll refer to the current example-program as the *enhanced DMA-based driver*, or simply, *the enhanced driver*. We'll refer to the prior example-program (`Src/mainUartDMABuff.c`) as the *basic DMA-based driver*, or simply, *the basic driver*.

The enhanced DMA-based driver is similar to the basic DMA-based driver. To summarize the enhanced driver: UART4 continuously sends data to USART2, at 9,600 baud. The driver is implemented using an ISR, and the ISR is run via a DMA interrupt. The ISR gets a buffer of data that the DMA controller received from USART2. The ISR puts the data in a stream-buffer. A task receives data from the stream-buffer, and prints it to SystemView.

Introducing stream buffers

The easiest way to explain stream-buffers is to show examples of their use, from our current example-program, the enhanced DMA-based driver

With the enhanced DMA-based driver, the ISR calls a FreeRTOS stream-buffer API, which copies the memory-buffer into a stream-buffer. This is similar to sending data to a FreeRTOS queue. A task is used to receive data from the stream-buffer and print the data. The task calls a stream-buffer API to copy data from the stream-buffer. This is similar to receiving data from a FreeRTOS queue.

Creating a stream-buffer

The stream-buffer itself is created in `main()`, by calling `xStreamBufferCreate`, as shown below. The second parameter specifies the stream-buffer's **trigger-level**, which is explained later.

```
#define DMA_BUFFER_LENGTH 16
#define STREAM_BUFFER_SIZE  (10 * DMA_BUFFER_LENGTH)
#define MIN_RECEIVE_SIZE  DMA_BUFFER_LENGTH
streamBuffer = xStreamBufferCreate( STREAM_BUFFER_SIZE, MIN_RECEIVE_SIZE);
```

The API's arguments and return value are:

- `STREAM_BUFFER_SIZE` : the size of the stream-buffer, in bytes.
- `MIN_RECEIVE_SIZE` : the *trigger-level* for receiving data from the stream-buffer.
- `streamBuffer` : the stream-buffer's handle.

Sending data to a stream-buffer

The DMA ISR is `DMA1_Stream5_IRQHandler`, and it is shown later. It sends data to the stream-buffer by calling `xStreamBufferSendFromISR`, as shown below. The API is for use in an ISR, so it does not block if the stream-buffer is full.

```
numWritten = xStreamBufferSendFromISR( streamBuffer, (uint8_t*) dmaBufferPtr,
                                       DMA_BUFFER_LENGTH,
                                       &xHigherPriorityTaskWoken);
```

The API's arguments and return value are:

- `streamBuffer` : the stream-buffer's handle.
- `dmaBufferPtr` : pointer to the data to copy into the stream-buffer.
- `DMA_BUFFER_LENGTH` : the maximum number of bytes to copy.
- `&xHigherPriorityTaskWoken` : if a task is blocked waiting on the stream-buffer, informs the scheduler whether the task should leave the blocked state.
- `numWritten` : the number of bytes that were copied to the stream-buffer. It can be less than `DMA_BUFFER_LENGTH`. If the stream-buffer is full, `0` is returned.

Receiving data from a stream-buffer

The task that receives data from the stream-buffer is `uartPrintOutTask`. It receives data by calling `xStreamBufferReceive`, as shown below.

```
#define DMA_BUFFER_LENGTH 16
#define MAX_BLOCK_TIME 100
numBytes = xStreamBufferReceive( streamBuffer, rxBufferedData,
                                 DMA_BUFFER_LENGTH, MAX_BLOCK_TIME );
```

The API's arguments and return value are:

- `streamBuffer` : the stream-buffer's handle
- `rxBufferedData` : pointer to the buffer to hold the received data
- `DMA_BUFFER_LENGTH` : the maximum number of bytes to receive
- `MAX_BLOCK_TIME`: the maximum number of ticks to block, waiting for data to arrive
- `numBytes` : the number of bytes received

When `xStreamBufferReceive` is called and sufficient bytes are not available, the API can block to wait for data to arrive. The algorithm for returning bytes after blocking is complex, and it uses the **trigger-level** specified in `xStreamBufferCreate`. The algorithm is described below:

If the maximum block-time is zero, then whatever bytes are in the stream-buffer will be returned, up to the maximum requested.

If the maximum block-time is not zero, then the trigger-level determines what is returned:

- If the number of bytes in the stream-buffer is greater than or equal to the trigger-level, then those bytes will be returned, up to the maximum requested. No blocking will occur.
- If the number of bytes in the buffer is less than the trigger-level, then the task will block:
    - While blocked, if the number of bytes in the buffer becomes greater than or equal to the trigger-level, then whatever bytes are in the stream-buffer will be returned, up to the maximum requested.
    - If the maximum block-time is reached, then the number of bytes in the buffer are less than the trigger-level. Whatever bytes are in the stream-buffer will be returned, up to the maximum requested.

Stream-buffers allow bytes to be passed between an interrupt service-routine and a task, or between one task and another task. Stream-buffers can be of arbitrary length, and any number of bytes can be read or written. For stream-buffer APIs, data is passed by copy. The data is copied into the buffer when sent, and copied out of the buffer when received.

An advantage of stream-buffers is that they are relatively fast, as FreeRTOS implements them internally by using task notifications. Recall that task-notifications are significantly faster than a FreeRTOS queue or semaphore.

Stream-buffers have limitations due to being implemented by task notifications. For example, if a stream-buffer API puts the calling task into the Blocked state, the API can potentially change the task's task-notification state and value. (For FreeRTOS versions that support indexed task-notifications, the stream-buffer implementation uses array index 0.)

Also, FreeRTOS's stream-buffer implementation assumes only one task or ISR writes to the buffer, and only one task or ISR reads from the buffer. For example, two tasks cannot concurrently send to the same stream-buffer. For two tasks to send to the same stream-buffer, their calls to `xStreamBufferSend` would have to be serialized, e.g., by using a semaphore or mutex. This way, only one task would call `xStreamBufferSend` at a time. Also, in this case, `xStreamBufferSend` would be called with a blocking time of 0, presumably to avoid changing the calling task's notification state and value. The same serialization techniques would be used for two tasks to receive from the same stream-buffer.

A new type of stream-buffer was introduced in FreeRTOS v11.1.0 (released April 2024). It's a **stream batching buffer**. The difference between a stream-buffer and a stream-batching-buffer is when a task reads from a non-empty buffer. For a stream-buffer, a read from a non-empty buffer returns immediately, regardless of the amount of data in the buffer. For a stream-batching-buffer, a read from a non-empty buffer will block until the amount of data in the buffer exceeds the trigger level, or the block-time expires.

Stream-buffers provide the convenience of a queue-based system, but with greater speed, due to being implemented by task-notifications.

Introducing double-buffered DMA

To further speed up our enhanced driver, we'll use two additional DMA features: double-buffering, and continuous reception of data. We'll explain how these features work, and how they are implemented in the example program.

In the basic DMA-based driver (`mainUartDMABuff.c`), DMA was used to fill a buffer. When the buffer filled, an interrupt was generated, and an ISR ran. To fill the buffer again, the DMA stream had to be restarted. In the current example-program, the DMA stream will run continuously, and not need to be restarted every time a buffer fills.

In addition, we'll configure DMA to use two buffers, as shown in the following figure. We'll name the buffers `rxData1` and `rxData2`. The figure shows `rxData1` being filled by the DMA controller.

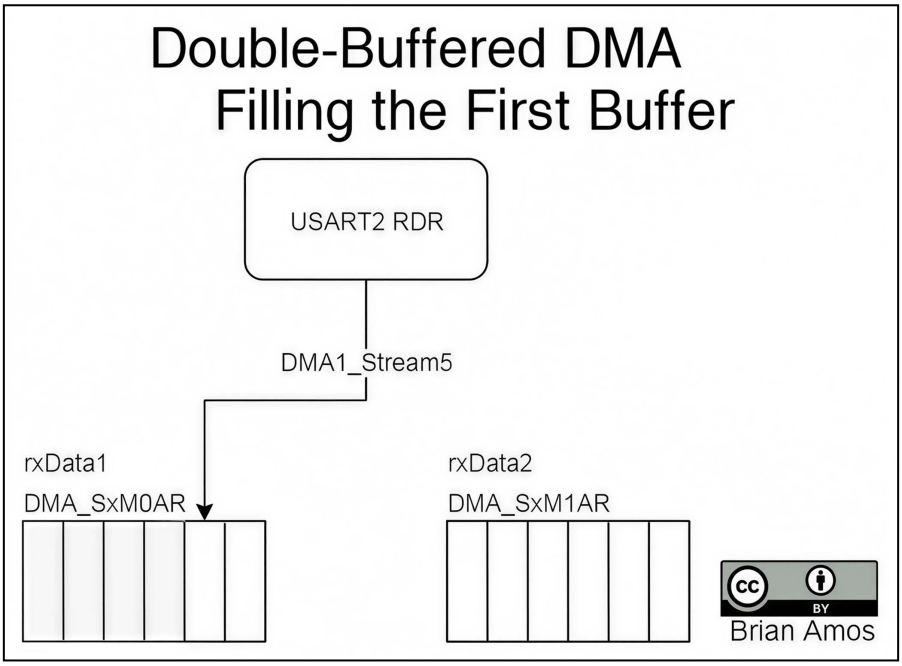


Figure 11.6 : Filling the first buffer of a double-buffered DMA

When `rxData1` is filled, an interrupt is generated, and the DMA controller will start filling `rxData2`, as shown in the following figure. The interrupt's ISR will write `rxData1` to a stream-buffer. When `rxData2` is filled, the process repeats: an interrupt is generated, the ISR writes `rxData2` to the stream-buffer, and the DMA controller will start filling `rxData1`. The DMA controller switches between the buffers automatically, based on how the controller is configured. So, the two buffers will be alternately filled and processed, indefinitely. UART4 sends data continuously, so there is a steady stream of bytes to receive.

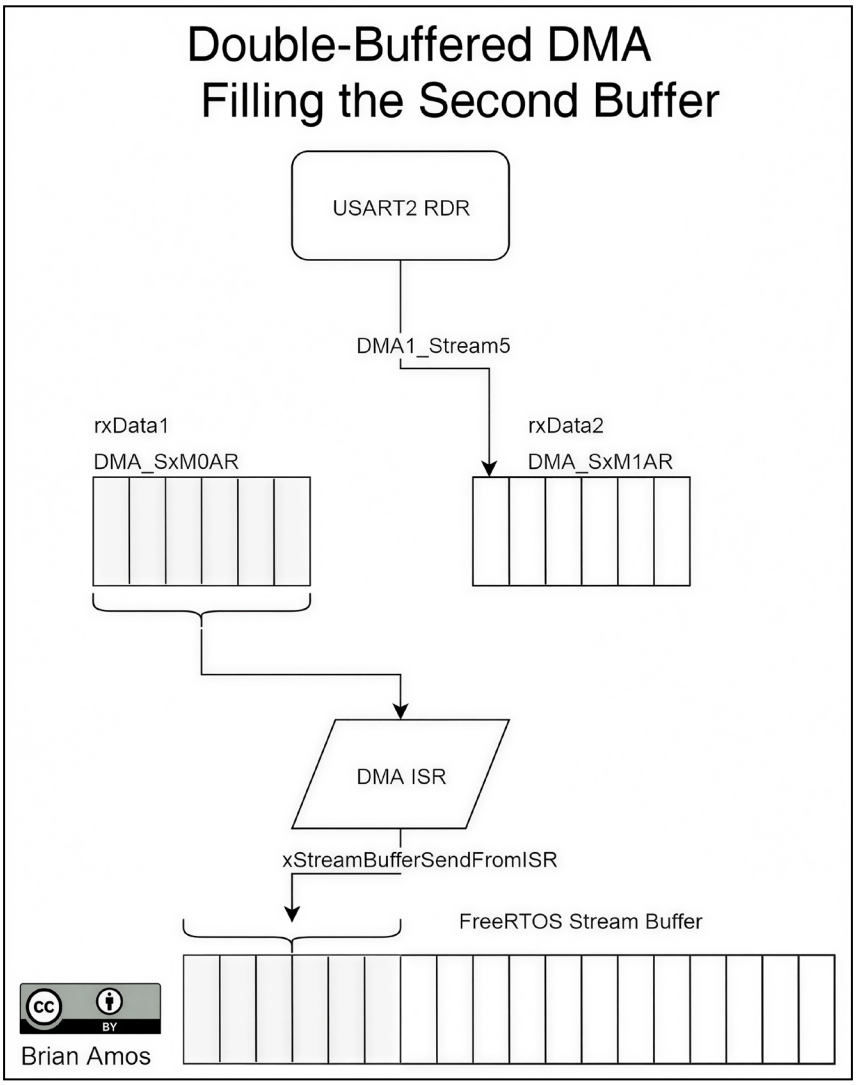


Figure 11.7 : Filling the second buffer of the double-buffered DMA

For the ISR, it is important to note that `xStreamBufferSendFromISR()` will add a *copy* of the data to the stream-buffer, not a reference. So, once `rxData1` has been filled, the ISR's call to `xStreamBufferSendFromISR()` must complete before `rxData2` has been filled. Otherwise, data arriving at USART2 will be lost. (This is an example of a *timing constraint*, which we discussed earlier in the book.)

With DMA, using two buffers like this is referred to as **double-buffer mode**. It is documented in the MCU's reference manual (RM0410), in section 8.3.10, "Double-buffer mode".



Even if you're programming for an MCU without an explicit **double-buffer mode**, most DMA controllers will have a **circular** mode with **half-transfer** and **full-transfer** interrupts. In this case, the same functionality can be achieved by generating an interrupt after each half of the buffer is filled.

The two prior example-programs implement buffer-based drivers. The programs are interrupt-driven, and use a USART ISR and a DMA ISR, respectively. However, a single buffer is used. In their performance analysis, we discussed the timing requirements for processing the buffer once it is filled. In particular, the filled-buffer has to be processed (e.g., printed) before the next byte arrives. Otherwise, data loss will occur. Using a single-buffer limits the driver's baud-rate. Using a single-buffer is also problematic, as it is not very resilient to variation in the baud-rate or to variation in the time to process the buffer. For the current example-program, we'll see how using two buffers can provide a much higher baud-rate, and greater resilience to variation.

Implementing the driver

The following sections describe how the enhanced DMA-based driver is implemented.

Configuring and starting DMA

DMA configuration and initialization are performed in `setupUSART2DMA()`. The same function was in the basic DMA-based driver, except now, it specifies the DMA mode as "circular," as shown below. So, DMA transfers are automatically restarted after a DMA buffer is filled. This allows continuous reception of data from USART2.

```
usart2DmaRx.Init.Mode = DMA_CIRCULAR;
```

The code to start the DMA receive is in `startReceiveDMA()`. The same function was in the basic DMA-based driver, except code was added to configure DMA in double-buffer mode, and to start double-buffer mode. The added code is described below.

In `startReceiveDMA()`, the secondary DMA buffer (`rxData2`) is set up by writing its address to the register `DMA1_Stream5->M1AR`, as shown below. (Some casting is required to keep the compiler from complaining too loudly that we're writing a pointer to a 32-bit memory address.):

```
// * Configure and initialize DMA stream.
// * Enable the DMA transfer-complete interrupt.
setupUSART2DMA();

// Setup second address for double-buffer mode
DMA1_Stream5->M1AR = (uint32_t) rxData2;
```

Also in `startReceiveDMA()`, the DMA transfer is started by calling `HAL_DMA_Start()`, as shown below. This HAL function will also set up the first DMA buffer (`rxData1`), by assigning its address to `DMA1_Stream5->M0AR`. The address of `rxData1` is passed to `HAL_DMA_Start()`.

Interestingly enough, STM HAL doesn't support double-buffer mode directly. In fact, calls to `HAL_DMA_Start` explicitly disable the mode. (See `DMA_SetConfig()` in `Drivers/STM32F7xx_HAL_Driver/Src/stm32f7xx_hal_dma.c`.) So, some manual setup with registers is required, as shown below (after letting `HAL` take care of most of the leg work in `HAL_DMA_Start()`):

```
// HAL_DMA_Start : Starts the DMA Transfer
if(HAL_DMA_Start(&usart2DmaRx, (uint32_t)&(USART2->RDR),
                 (uint32_t)rxData1, DMA_BUFFER_LENGTH) != HAL_OK)
{
    . . .
    return -1;
}
// Disable the stream and controller so we can setup dual buffers
__HAL_DMA_DISABLE(&usart2DmaRx);
// Wait for the EN bit to be set to 0.
while((DMA1_Stream5->CR & DMA_SxCR_EN) != 0);

// Set the double-buffer mode
DMA1_Stream5->CR |= DMA_SxCR_DBM;

// Re-enable the stream and controller
__HAL_DMA_ENABLE(&usart2DmaRx);
DMA1_Stream5->CR |= DMA_SxCR_EN;
```

The example-program's task is `uartPrintOutTask` (shown below). The task configures and starts DMA and USART2, before the task's infinite while-loop. The DMA-receive is initialized and started by calling `startReceiveDMA()`. In the task, USART2 is initialized by calling `STM_UartInit()`, and USART2 is started by setting its control registers (as in the basic DMA-based driver).

Receiving data from the stream-buffer

The task calls `xStreamBufferReceive` to receive data from the stream-buffer:

```
void uartPrintOutTask( void* NotUsed)
{
    #define MAX_BLOCK_TIME 100
    uint8_t rxBufferedData[DMA_BUFFER_LENGTH+1];
    . . .

    // Configure and initialize the DMA stream
    startReceiveDMA();
    // Initialize USART2, pass usart2DmaRx for use in receive
    STM_UartInit(USART2, BAUDRATE, NULL, &usart2DmaRx);
    // DMA mode is enabled for reception
    USART2->CR3 |= USART_CR3_DMAR_Msk;

    // Clear error flags.
    USART2->ICR |= ( USART_ICR_FECF | USART_ICR_PECF |
                     USART_ICR_NCF | USART_ICR_ORECF );
    // Enable the UART
    USART2->CR1 |= USART_CR1_UE;

    while(1)
    {
        uint8_t numBytes = xStreamBufferReceive( streamBuffer,
                                                 rxBufferedData,
                                                 DMA_BUFFER_LENGTH,
                                                 MAX_BLOCK_TIME );

        if(numBytes > 0)
        {
            // Code not shown for setting null-terminators in the buffer.
            . . .
```

```
            SEGGER_SYSVIEW_PrintfHost("received: ");
            SEGGER_SYSVIEW_Print((char*)rxBufferedData);
            . . .
        }
        else
        {
            SEGGER_SYSVIEW_PrintfHost("timeout");
        }
    } // End of: while(1)
}
```

In the preceding code, for the call to `xStreamBufferReceive`, note the following about the arguments and the return value:

- `streamBuffer`: The stream-buffer's handle.
- `rxBufferedData`: The buffer that bytes are copied into. It can hold the 16-byte string sent by UART4.
- `DMA_BUFFER_LENGTH`: The maximum number of bytes that will be copied into `rxBufferedData`.
- `MAX_BLOCK_TIME`: The timeout, 100 ticks.
- `numBytes`: the number of bytes returned.

If the call to `xStreamBufferReceive()` blocks, it will return after at least the trigger-level number of bytes are available or 100 ticks have elapsed. The trigger-level is specified when the stream-buffer was created. It is 16. The algorithm for returning bytes after blocking was described earlier.

Now that we have a good idea of what the stream-buffer's receiving side looks like, let's look at some of the details of the ISR, and the stream-buffer's sending side.

Populating the stream-buffer

The stream-buffer will be populated from inside the DMA ISR:

```
void DMA1_Stream5_IRQHandler(void)
{
    . . .

    if(DMA1->HISR & DMA_HISR_TCIF5)
    {
        // Copy data from the correct buffer into the FreeRTOS stream buffer.
```

```
        // * The DMA_SxCR_CT bit indicates the "Current Target",
        //   which is either DMA_SxM0AR or DMA_SxM1AR.
        if(DMA1_Stream5->CR & DMA_SxCR_CT)
        {
            dmaBufferPtr = rxData1;
        }
        else
        {
            dmaBufferPtr = rxData2;
        }

        // Send the buffer to the stream-buffer.
        numWritten = xStreamBufferSendFromISR( streamBuffer,
                                               (uint8_t*) dmaBufferPtr,
                                               DMA_BUFFER_LENGTH,
                                               &xHigherPriorityTaskWoken);

        // OH NO - not everything was written to the stream
        // Spin, to identify the error.
        while(numWritten != DMA_BUFFER_LENGTH);

        // DMA high interrupt flag clear register (DMA_HIFCR)
        // * CTCIF[7:4]: stream x clear transfer-complete interrupt-flag
        //   * Writing 1 to this bit clears the corresponding TCIFx flag
        //     in the DMA_HISR register.
        DMA1->HIFCR |= DMA_HIFCR_CTCIF5;
    }
    . . .
```

Here are some of the more noteworthy items in this ISR:

- `if(DMA1->HISR & DMA_HISR_TCIF5)`: Checks the transmit complete flag. `DMA_HISR_TCIF5` guarantees that a transfer has indeed completed (rather than entering the ISR because of an error).
- `(DMA1_Stream5->CR & DMA_SxCR_CT)`: Checks the current target bit. This bit indicates which buffer (`DMA_SxM0AR` or `DMA_SxM1AR`) is currently being *filled* by the DMA controller. We will use the other buffer, and copy its data into the stream-buffer.

- The call to `xStreamBufferSendFromISR` pushes the entirety of `rxData1` or `rxData2` (each of length `DMA_BUFFER_LENGTH`) into `streamBuffer` in one go.

A few things to remember are as follows:

- Data is being transferred to the stream by value (not reference). That is, FreeRTOS is using `memcpy` to make a copy of all of the data moving into the stream-buffer (and again when removing data). The larger the buffer, the more time it will take to copy – additional RAM will also be used.
- Instead of performing the copy inside the interrupt, under certain circumstances, it may be preferable to signal a semaphore or task notification and perform the copy in a high-priority task instead – for example, if a large buffer is being filled. However, you'll need to guarantee that other interrupts don't starve the task performing the `xStreamBufferSend` or data will be lost.
- There are trade-offs when using DMA. Larger buffers mean fewer interruptions to transfer data, but they also mean an increase in latency. The larger the buffer, the longer the data will sit in the buffer before being processed.
- This implementation is only well suited to continuous data streams – if the data stream stops before filling a buffer, the last DMA transfer will never complete.

This approach to pushing received data from a peripheral into memory works very well when data is continuously streaming. It can also work extremely well for the reception of messages with a known number of bytes (when there are no transmission errors). However, there are some ways to improve it, which will be shown later.

Analyzing the performance

Let's take a look at the performance of this enhanced DMA-based driver compared to the basic DMA-based driver. The main differences are that the enhanced driver uses two 16-byte buffers, versus one 16-byte buffer. Also the enhanced driver uses stream-buffers, versus a semaphore. For both drivers, a 9,600 baud-rate is used.

The enhanced DMA-based driver was run, and the resultant SytemView **Timeline** window is shown in the following three figures. It looks very similar to the results from the basic DMA-based driver. It shows the ISR receives a buffer, and the task immediately prints it.

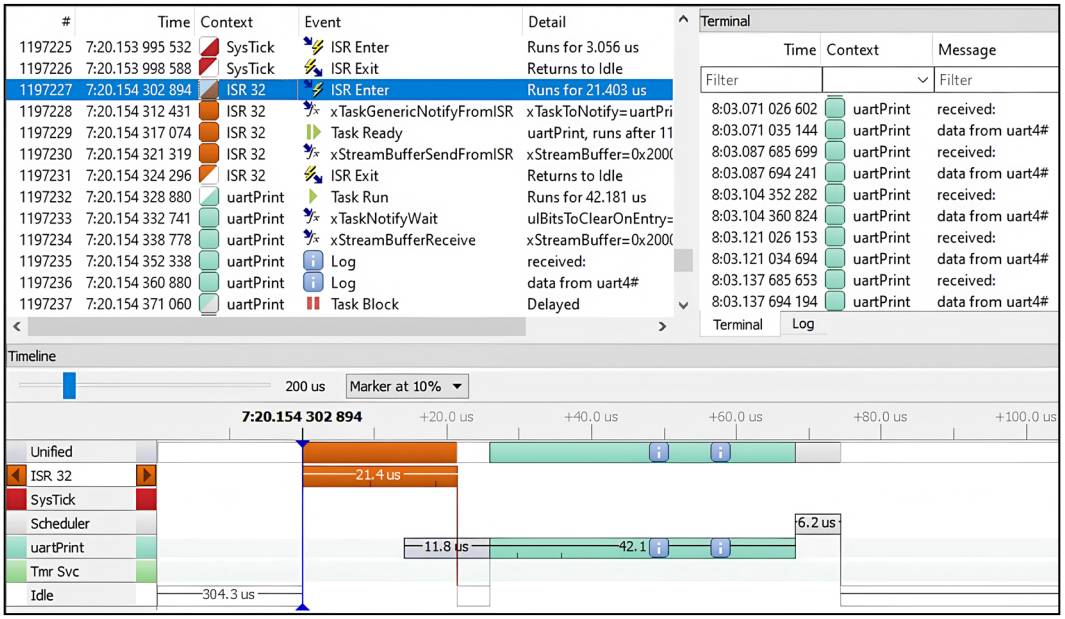


Figure 11.8 : SystemView

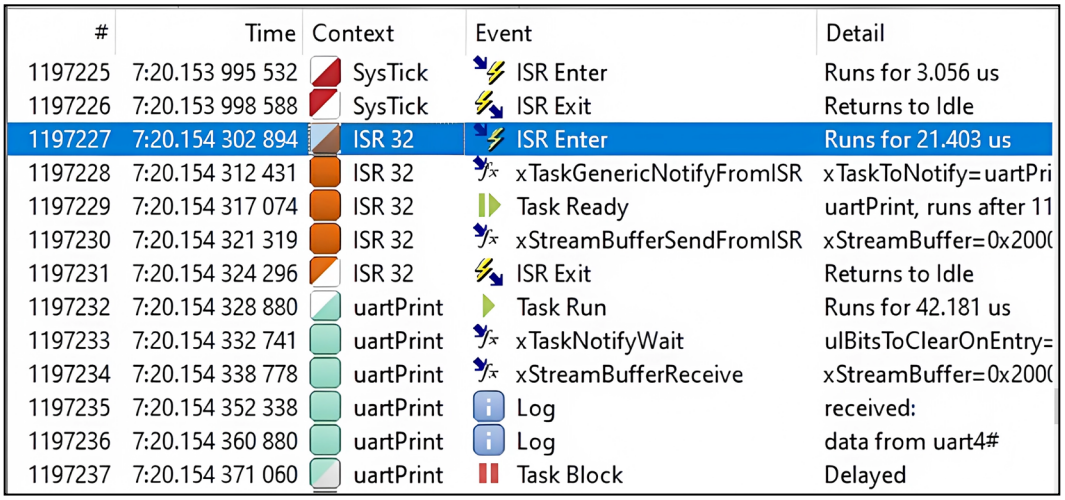


Figure 11.8a : The left side of figure 11.8

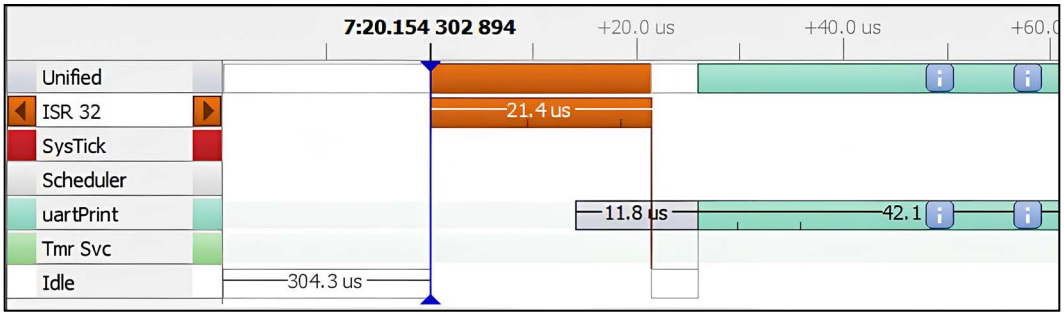


Figure 11.8b : The bottom of figure 11.8

The SystemView **Contexts** window is shown below, and an explanation is provided. The figure summarizes the system's performance after 8:03 minutes of operation.

Contexts

Name	Type	CPU Load	Activations	Max Run Time	Max Run Time
ISR 32	#32	0.13 %	28 995	0.020 981 ms	0.023 028 ms
SysTick	#15	0.31 %	483 239	0.003 046 ms	0.004 537 ms
Scheduler		0.04 %	28 995	0.005 944 ms	0.006 880 ms
uartPrint	@3	0.26 %	28 995	0.041 403 ms	0.047 463 ms
Tmr Svc	@2	0.00 %	0	0.000 000 ms	0.000 000 ms
Idle		99.26 %	28 995	16.536 699 ms	16.547 546 ms

Figure 11.9 : SystemView Contexts window

From the figure above, we can make the following observations about the system's performance, compared to the basic DMA-based driver:

- The DMA ISR is now consuming 0.13% of CPU cycles, versus 0.09% previously. It's a 44% increase.
- The ISR's frequency is 60 Hz (28,995 activations / 483 sec), the same as before.
- The task is now consuming 0.26% of CPU cycles, vs. 0.28% previously. It's a 7% decrease.
- The Idle time is 99.26%, versus 99.25% previously. They are essentially the same.

The two example-programs have very similar performance, overall. When using stream-buffers, the ISR takes longer due to copying a 16-byte buffer. In the basic DMA-based driver, the ISR just called a semaphore API, and the buffer was not copied by the ISR. In the enhanced DMA-based driver, its task takes slightly less time. The enhanced driver uses continuous DMA transfers, so the task does not need to restart the DMA stream as in the basic driver. Also, the two drivers call different FreeRTOS APIs: a stream-buffer API versus a semaphore API. For both drivers, the task copies a 16-byte buffer.

The primary performance difference between the two example-programs is due to using two DMA buffers, versus one buffer. Using two buffers provides a much higher upper-bound for the driver's baud-rate.

For the basic DMA-based driver, each time the DMA buffer was filled, the ISR and task had to process the buffer and restart the DMA transfer. To not lose data, this all had to occur within the amount of time it took for a byte to arrive. Earlier, we used our performance data to calculate an upper-bound for the basic driver's baud-rate, and it was 143,472 bits per second.

In the enhanced DMA-based driver, the ISR and task have to process the DMA buffer that was just filled, before the next DMA buffer is filled. That is, before 16 bytes arrive. This creates a much higher upper-bound for the driver's baud rate.

For the enhanced DMA-based driver, our performance data shows how long it takes for the ISR and task to process a DMA buffer. The ISR runs for at most 0.0230 ms, and the task runs for at most 0.0475 ms. From the **Timeline** window (not shown), the context switch between them is less than 0.0045 ms. So, the time to process the DMA buffer is at most 0.075 ms. This gives an upper-bound on the baud-rate of 2,133,333 bits per-second:

```
( (1/(0.075 * 0.001)) [buffer per second] * 160 [bits sent per buffer] )
```

By using DMA with two buffers, the upper-bound calculation for the baud-rate is 15 times greater than using one buffer. This increase can be explained by both the enhanced driver's increase in buffer size, and its ISR being slower due to the buffer-copy. In the basic driver, the DMA buffer has to be processed within the time of a byte arriving. In the enhanced driver, the DMA buffer has to be processed within the time of 16 bytes arriving.

Improving the stream-buffer

Our recent example-programs have used UART4 to send data to USART2. The data is sent continuously, and at a constant rate (9,600 baud). In practice, received data can be intermittent. This creates complications when using DMA to buffer the received data, especially if an interrupt is only generated when the buffer is filled. For instance, a buffer could get partially filled, and if no more data is sent, an interrupt would never be generated to indicate data was available.

In order to deal with an intermittent data stream, two possible approaches for this setup are:

- The USART peripheral on this MCU is capable of detecting an "idle line" and generating an interrupt by setting the bit `USART_CR1:IDLEE`.
- The USART peripheral also has a `receive` timeout that can generate an interrupt after no start bits have been detected for a specified number of bit times (0 to 16,777,215).
    - This timeout is specified in the register `USART_RTOR:RTO[23:0]`.
    - The feature can be enabled with `USART_CR2:RTOEN` and the interrupts can be enabled with `USART_CR1:RTOIE`.

Either of these features could be used to generate a USART interrupt, cut the DMA transfer short, and transfer the data to the stream-buffer.

For extremely high baud-rates, care needs to be taken when using the idle-line approach because the number of interrupts generated is only capped by the baud rate. If there is inter-character spacing (idle time between each character being sent), you'll wind up with an interrupt-driven approach (with even more overhead than normal).

On the other hand, using the `receive` timeout feature means additional latency before processing the incoming data. As usual, there is no *one-size-fits-all* solution here.

So, now that we've worked through a simple example of each driver type, let's look at how to choose the best driver-model for a given system.

Choosing a driver model

Selecting the *best* driver for a given system depends on several factors. There are many ways a driver can be structured, as we have been exploring in this chapter and the last. Figuring out the best model for your application will depend on several factors.

Questions to ask when choosing a driver model

Here are a few questions that you can ask that may help you determine the best model for your application:

- How is the driver's calling-code designed?
- How much delay is acceptable?
- How fast is data moving?
- What type of device are you interfacing?

Let's answer these questions one by one.

How is the driver's calling-code designed?

What is the intended design of the higher-level code using the driver? Will it operate on individual characters or bytes as they come in? Or does it make more sense for the higher-level code to batch-process transfers in blocks of bytes?

Queue-based drivers are very useful when dealing with unknown amounts (or streams) of data that can come in at any point in time. They are also a very natural fit for code that processes individual bytes – an earlier version of `uartPrintOutTask` was a good example of this:

```
while(1)
{
    xQueueReceive(uart2_BytesReceived, &nextByte, portMAX_DELAY);
   // Do something with the byte received
    SEGGER_SYSVIEW_PrintfHost("%c", nextByte);
}
```

While ring-buffer implementations (such as the one in the preceding code) are perfect for streamed data, other code naturally gravitates toward operating on blocks of data. Say, for example, one of the structures defined in an earlier chapter (`Chapter_09/Src/mainQueueCompositePassByValue.c`):

```
typedef struct
{
    uint8_t redLEDState : 1;
    uint8_t blueLEDState : 1;
    uint8_t greenLEDState : 1;
    uint32_t msDelayTime;
} LedStates_t;
```

Rather than operate on individual bytes, it is very convenient for the receiving side to pull in an instance of the entire struct at once. The following code is designed to receive an entire copy of `LedStates_t` from a queue. After the struct is received, it can be operated on by simply referencing members of the struct, such as checking `redLEDState`, in this example:

```
LedStates_t nextCmd;
while(1)
{
    xQueueReceive(ledCmdQueue, &nextCmd, portMAX_DELAY);

    if(nextCmd.redLEDState == 1)
        RedLed.On();
    Else
    ...
```

To pass a data structure over a serial communication line, it can be *serialized*. Our `LedStates_t` struct can be serialized as a block of 5 bytes. All three red, green, and blue state values can be packed into 3 bits of a byte and the `msDelayTime` will take 4 bytes:

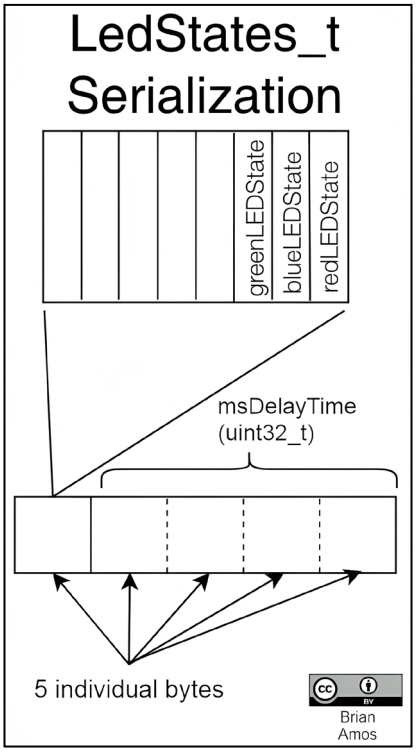


Figure 11.10 : Serializing the LedStates_t structs



Serialization is a broad topic in itself. There are trade-offs to be made for portability, ease of use, code fragility, and speed. A discussion on all of these points is outside the scope of this chapter. Details of endianness and the *best* way of serializing/deserializing this particular data structure have been purposely ignored in the diagram. The main takeaway is that the struct can be represented by a block of 5 bytes.

In this case, it makes sense for the underlying peripheral-driver to operate on a buffer of 5 bytes, so a buffer-based approach that groups a transfer into a block of 5 bytes is more natural than serially processing a stream of bytes. The following pseudo-code outlines an approach based on the buffer-based driver we showed earlier (the basic DMA-based driver):

```
uint8_t ledCmdBuff[5];
startReceiveDMA(ledCmdBuff, 5);
// Wait for reception to complete
xSemaphoreTake(cmdReceived, portMAX_DELAY);
// Populate an LED command with data received from the serial port
LedStates_t ledCmd = parseMsg(ledCmdBuff);
// Send the command to the queue
xQueueSend(ledCmdQueue, &ledCmd, portMAX_DELAY);
```

In this chapter, we have covered two different approaches that can provide efficient implementations:

- A buffer-based driver (receiving 5 bytes at a time)
- A stream-buffer-based driver (the receiving side can be configured to acquire 5 bytes at a time)



FreeRTOS *message buffers* could also be used instead of a stream-buffer to provide a more flexible solution. Message-buffers are built on top of stream-buffers, but have a more flexible blocking configuration. They allow different message sizes to be configured per `receive` call, so the same buffer can be used to group receptions into a size of 5 bytes (or any other desired size) each time `xMessageBufferReceive` is called. With stream-buffers, the message size is rigidly defined when creating the stream-buffer by setting the `xTriggerLevelBytes` parameter in `xStreamBufferCreate`. Unlike stream-buffers, message-buffers will only return full messages, not individual bytes.

How much delay is acceptable?

Depending on the exact function being implemented, minimal delay may be desired. In this case, buffer-based implementations can sometimes have a slight advantage. They allow the calling code (e.g., in a task) to be set up as extremely high priority, without causing significant context switching in the rest of the application.

With a buffer-based setup, after the last byte of a message is transferred, the task will be notified and immediately run. This is better than having the high-priority task perform byte-wise parsing of the message since it will be interrupting other tasks continually each time a byte is received. In a byte-wise queue-based approach, the task waiting on the queue would need to be set to a very high priority if the incoming message was extremely important. This combination causes quite a bit of task context-switching versus a buffer approach, which only has a single semaphore (or direct task notification) when the transfer is finished.

Sometimes, timing constraints are so tight neither queues nor an entire block transfer may be acceptable (bytes might need to be processed as they come in). This approach will sometimes eliminate the need for intermediate buffers as well. A fully custom ISR can be written in these cases, but it won't be easily reused.

Try to avoid lumping *business logic* (application-level logic not immediately required for servicing the peripheral) into ISRs whenever possible. It complicates testing and reduces code reuse. After a few months (or years) of writing code like this, you'll likely notice that you've got dozens of ISRs that look *almost* the same but behave in subtly different ways.

How fast is data moving?

While extremely convenient, queues are a fairly expensive way to pass individual bytes around a system, i.e., they are expensive in consuming CPU resources, code space, data memory, and bandwidth. Even an interrupt-based driver has limitations in regard to being able to process incoming data in time to meet deadlines. Our example used a meager 9,600 baud transfer. Individual characters were transferred into the queue within 40 µs of being received, but what happens if the baud rate is 115,200 baud? Now, instead of having around 1 character per millisecond, each character would need to be added to the queue in less than 9 µs. A driver that takes 40 µs per interrupt isn't going to be acceptable here, so using a simple queue approach isn't a viable option.

We saw that the stream-buffer implementation with DMA was a viable solution in place of a queue. Using some type of double-buffering technique for high-speed, continuous streams of data is critical. This becomes an especially convenient technique when coupled with a highly efficient RTOS primitive, such as stream-buffers or message-buffers.

Interrupts and DMA-based drivers that move data directly into a *raw* memory buffer are also quite viable when speeds are high, but they don't have the convenience of a queue-like interface.

What type of device are you interfacing?

Some peripherals and external devices will naturally lean toward one implementation or another. When receiving asynchronous data, queues are a fairly natural choice because they provide an easy mechanism for constantly capturing incoming data. UARTs, USB virtual comms, network streams, and timer captures are all very naturally implemented with a byte-wise queue implementation (at least at the lowest level).

Synchronous-based devices, such as a serial peripheral interface (SPI) and Inter-Integrated Circuit (I2C), are easily implemented with block-based transfers on the master side since the number of bytes is known ahead of time (the master needs to supply the clock signal for both bytes-sent and bytes-received).

Rules of thumb for choosing a driver model

Now that you understand some of the questions that we should ask when trying to determine which driver model to use, here are some rules of thumb to help answer those questions.

When to use queue-based drivers

Here are some cases where it is advantageous to use a queue as the interface of a driver:

- When the peripheral/application needs to receive data of an unknown length
- When data must be received asynchronously to requests
- When a driver should receive data from multiple sources without blocking the caller
- When data rates are sufficiently slow to allow a minimum of 10's of µS per interrupt (when being implemented on the hardware, in this example)

When to use buffer-based drivers

Some cases where raw buffer-based drivers are extremely useful are as follows:

- When large buffers are required because large amounts of data will be received at once
- During transaction-based communication protocols, especially when the length of the received data is known beforehand

When to use stream-buffers

Some cases where stream-buffer drivers have advantages:

- In places where a raw buffer would be used but with the added benefit of providing an efficient queue API.
- In place of a raw buffer, where a *little bit* of extra overhead may be tolerated.

These are just some of the considerations you'll likely face when creating drivers. The considerations are mainly aimed at communication peripherals, since that's what our examples covered. There are also some considerations to be made when choosing to use third-party libraries and drivers, as we'll see in the next section.

Using third-party libraries and drivers

If you've been following along closely, you may have noticed a few things:

- STM HAL (the vendor-supplied hardware abstraction layer) is used for initial peripheral configuration (e.g., `HAL_DMA_Init`). This is because HAL does a very good job of making peripheral configuration easy. In addition to using HAL, it is also extremely helpful to use tools such as STM32CubeIDE to generate some boilerplate code, and use it as a point of reference when first interacting with a new chip.
- When it is time to implement details of interrupt-driven transactions, we've been making a lot of calls directly to MCU peripheral registers, rather than letting HAL manage all transactions for us. There were a couple of reasons for this:
    - We wanted to be closer to the hardware to get a better understanding of how things were really working in the system.
    - Some of the setups weren't directly supported by HAL, such as DMA double-buffering.

In general, you should use as much vendor-supplied code as you (or your project/company) are comfortable with. If the code is well written and works reliably, then there *usually* aren't too many arguments for *not* using it.

That being said, here are some potential issues when using vendor-supplied drivers:

- They may use polling instead of interrupts or DMA.
- Tying into the driver's interrupts may be cumbersome or inflexible.
- There is potentially *lots* of extra overhead since many chips and use-cases are likely covered by drivers (they need to solve *everyone's* problems, not just yours).

- It might take longer to fully grasp and understand a complex API than working directly with the peripheral hardware (for simple peripherals).
- Your code inherits any bugs in the vendor code. You may find that you have to modify the vendor code to work properly. Then you may find yourself maintaining a variant of that code throughout the lifetime of the code-base.

The following are examples of when to write **bare-metal drivers**:

- When you need to understand every nuance of the code for meeting internal quality-standards or regulatory requirements
- When a vendor's driver is broken or buggy
- When speed matters
- When an exotic configuration is required
- As a learning exercise

Ideally, transitioning between third-party drivers and your own drivers would be perfectly seamless. If it isn't, it means that the higher-level code is tightly coupled to the hardware. This tight coupling is perfectly acceptable for sufficiently small *one-off* and *throw-away* projects, but if you're attempting to develop a code-base for the long term, investing in creating a loosely coupled architecture will pay dividends. Having **loose coupling** (eliminating dependencies between the exact driver implementation and higher-level code) also provides flexibility in the implementation of the individual components. Loose coupling ensures transitioning between custom drivers and third-party drivers doesn't necessitate a major rewrite of high-level code. Loose coupling also makes it possible to test small portions of the code-base in isolation.

Summary

In this chapter, we looked at ways to improve our UART driver. We first used DMA and a single buffer to improve the efficiency of our ISR-based driver. Next, we further improved the driver by using DMA with two buffers, continuous DMA reception, and a stream-buffer to send data from an ISR to a task. Performance analysis of both approaches showed that using DMA with two buffers allowed more time for processing the buffer, and thereby provided a much higher upper-bound for the baud-rate.

Considerations for choosing between driver implementation-options was also discussed, as well as when it is appropriate to use third-party peripheral drivers (STM HAL) and when "rolling your own" is best.

Our exploration of drivers and ISRs was really just the tip of the iceberg when it comes to driver implementation. There are many other approaches and techniques that can be used when creating efficient driver implementations, including additional RTOS primitives, and configuring MCU-specific functionality. Your designs don't need to be limited by what happens to be provided by a vendor.

You should now have a good understanding of several ways that low-level drivers can be implemented. In the next chapter, we'll take a look at how these drivers can be safely presented to higher-level code across multiple tasks. Providing easy access to drivers makes developing the final application fast and flexible.

Further reading

- DMA application-notes from ST: *Using the STM32F2, STM32F4 and STM32F7 Series DMA controller*, Document ID: AN4031.
- DMA presentation slides from ST: *STM32F7 - DMA : Direct memory access controller*, `https://www.st.com/resource/en/product_training/STM32F7_System_DMA.pdf`
- Stream-buffer tutorial from FreeRTOS.og, `https://www.freertos.org/RTOS-stream-buffer-example.html`

12

Sharing Hardware Peripherals among Tasks

This chapter shows how FreeRTOS can be used to implement a driver that is shared among multiple tasks. Two drivers are created for sending messages over USB. The messages are sent from the dev-board to the development-computer. The first driver supports one task sending USB messages. The second driver enhances the first driver, by supporting multiple tasks sending USB messages. The chapter focuses on FreeRTOS features that are helpful in creating drivers. The features used are ones we've worked with earlier: tasks, stream-buffers, semaphores, mutexes, and timers.

The chapter has three example-programs. They are in the IDE-project `Chapter_12`. The first example-program introduces sending messages over USB, and it shows the need for a driver for sending USB messages. The next two example-programs implement the two drivers, respectively, along with programs that call the drivers.

The dev-board's MCU has a USB peripheral. The USB peripheral, and it's libraries, are especially complex, so our description of USB is limited to the USB-use in the example-programs and drivers. Each driver provides a C-function that tasks can call to send a message over USB. The function sends the message to a FreeRTOS stream-buffer. Each driver also has a task that receives data from the stream-buffer, and then calls USB APIs to send the data to the development-computer.

The USB peripheral is connected to a Virtual COM Port (VCP) on the development-computer (a PC).

Data is sent over the USB-connection to the VCP. PC applications can use the VCP as if it was a standard COM-port for a serial interface, such as RS-232. On the PC, we'll connect a serial-console app to the VCP, to display the data that is sent over the USB-connection. In general, using a VCP like this makes it possible to use existing applications on a PC that were made to work with COM ports. This is often much simpler than trying to write custom PC-software to interface with a custom USB-device.

To run the example-programs on the dev-board, two USB cables are needed. And, a serial-console app is needed on the development-computer. In this chapter, the serial-console app that is shown is Tera Term. It is open-source and runs on Windows. (Tera Term version 4.106 is shown here. Presently, the current version is 5.4.0, and it works similarly.) Any similar serial-console app should work as well (e.g., CoolTerm or minicom).

The chapter contains the following sections:

- Understanding how messages are sent over USB
- Sending messages using a simple USB API
- Creating a VCP driver, using a stream-buffer
- Enhancing the VCP driver, for use by multiple tasks

The first section shows the hardware-configuration used in the chapter. It also describes how USB works in sending messages. The remaining three sections describe the three example-programs, respectively.

Understanding how messages are sent over USB

The chapter's example-programs use the hardware configuration shown in the diagram below. Two USB cables connect the dev-board to the development computer. One is the **J-Link USB-cable**, which we've been using all along. The other is the **User USB-cable**, which we'll use to send messages from an example-program on the dev-board, to the development-computer.

The user USB-cable is plugged into the dev-board's **User USB-connector**, and the other end of the cable is plugged into the development-computer, in one of its USB connectors. The diagram will be used in explaining how messages are sent over USB. The diagram and explanation are simplified models of how USB works here, so much is omitted, and parts are approximations.

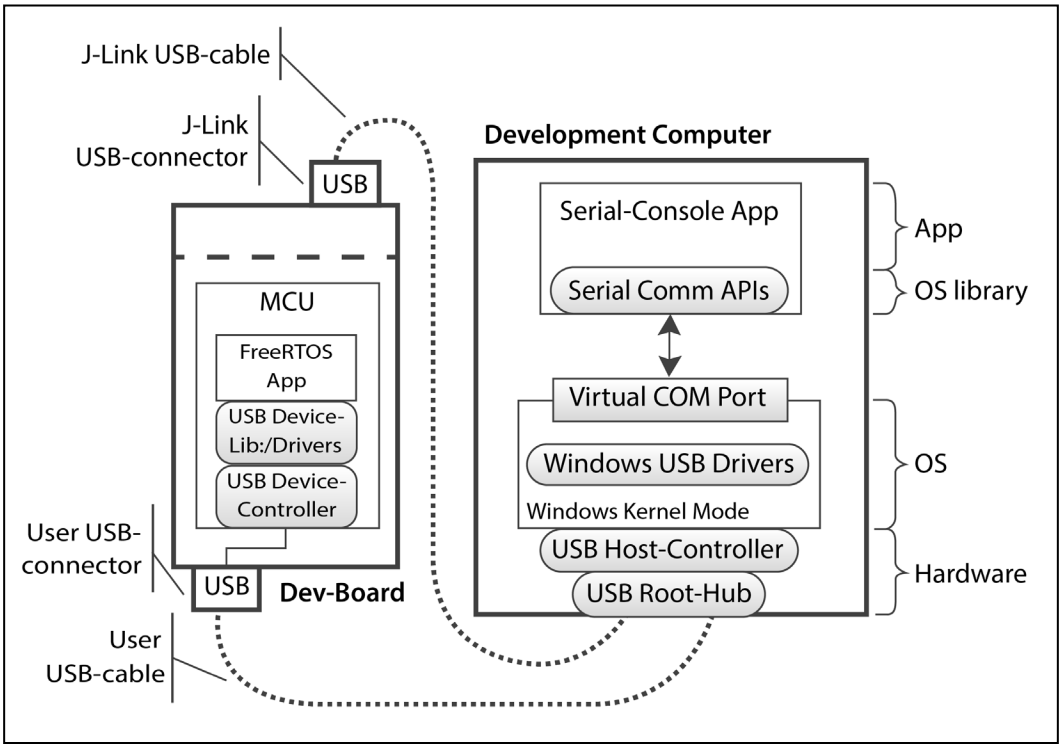


Figure 12.1 : Using USB to send messages

In the preceding diagram, the **FreeRTOS App** is a program, and in our use, it can be any of the chapter's example-programs. The example-program sends a message from the dev-board to the development-computer, over the user USB-cable. The messages are displayed on the development-computer, on a serial-console. The serial-console used in this chapter is **Tera Term**. It is shown here, with example messages:

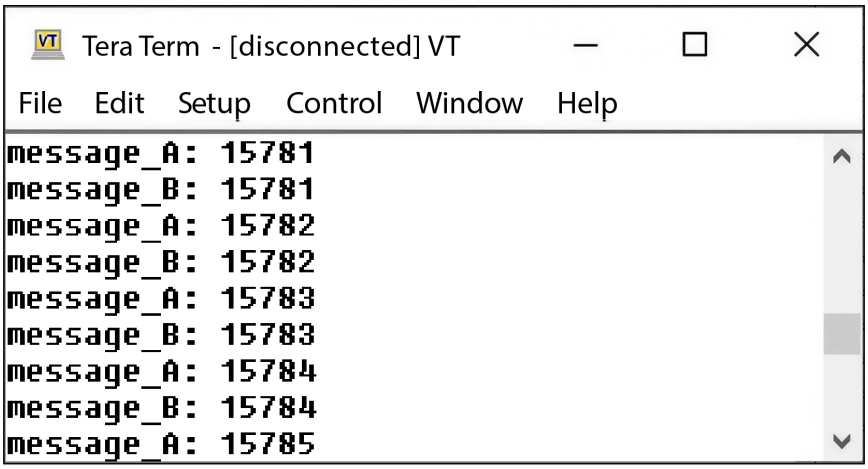


Figure 12.2 : The serial-console, Tera Term, displaying messages sent over USB

We'll describe the path a message takes, starting at an example-program running on the MCU, and ending at the serial-console running on the development-computer.

An example-program sends a message over USB by calling APIs from the **STM32Cube USB Device-Library**. An example API is `CDC_Transmit_FS()`, which is called to send a message. Examples of other APIs from the library include those used to initialize the USB device and those used to create the USB connection to the development-computer.

The example-programs use various APIs from the USB device-library, and those APIs will be described in the upcoming sections. The chapter's focus is on using FreeRTOS to create drivers, and not USB. So, those APIs are presented as black-boxes. A more in-depth description of USB and the USB device-library is beyond the scope of the book. Additional sources on USB are provided at the end of the chapter.

When a USB device-library API is called to send a message, functions in the library will transfer the message to the **USB Device-Controller**. The **USB device-controller** is a hardware component that resides on the MCU.

As shown in *Figure 12.1*, the development-computer has a **USB Host-Controller**, which is also a hardware component. The host-controller and device-controller communicate with each other. The USB device-library has functions and drivers that allow the device-controller to communicate with the host-controller. This includes initializing the device-controller, sending messages to the host-controller, etc. The host-controller is the master component, and the device-controller is a slave.

Also shown in *Figure 12.1*, when the host-controller receives a message from the device-controller, the host-controller passes the message to the **Windows USB-drivers**. And, they pass the message onto the **Virtual COM Port (VCP)**. The VCP is a virtual serial-port.

The **Serial-Console App** connects to the VCP, just like it would connect to a standard COM-port for a serial-interface (e.g., RS-232). However, with the VCP, there's no serial-interface, e.g., there's no RS-232 interface.

Referring to *Figure 12.1* again, a serial-console app receives data from a serial-port by calling "Serial-Comm APIs". For example, the Win32 API library has APIs for serial-communication, such as `WaitCommEvent()` and `ReadFile()`. In our case, the serial-communication APIs receive data from the VCP. The VCP passes data from the USB connection, to the serial-app.

For Windows apps that use a serial-port, VCP makes it possible for them to send and receive data over USB. That is the purpose of VCP. Modern computers typically do not have serial-ports, but almost all have USB ports. For legacy Windows-apps that use a serial-port, VCP makes it possible to use the app on computers with a USB port, but no serial-port. Similarly, VCP can be useful when creating a device that sends data over USB to a Windows computer. VCP makes it possible to send the data to an existing Windows app that uses a serial-port, such as a serial-console app. This can avoid having to create a Windows app to interface with the device.

This chapter's example-programs send data to the serial-console app, over USB. The example-programs could also receive data from the serial-console app over USB, but that is not implemented.

When the dev-board is connected to the development-computer over USB, the USB connection's VCP is shown in Windows' Device Manager. This is illustrated below. For the example-programs, the VCP will be listed as **USB Serial Device**. In the figure below, the VCP is **COM4**.

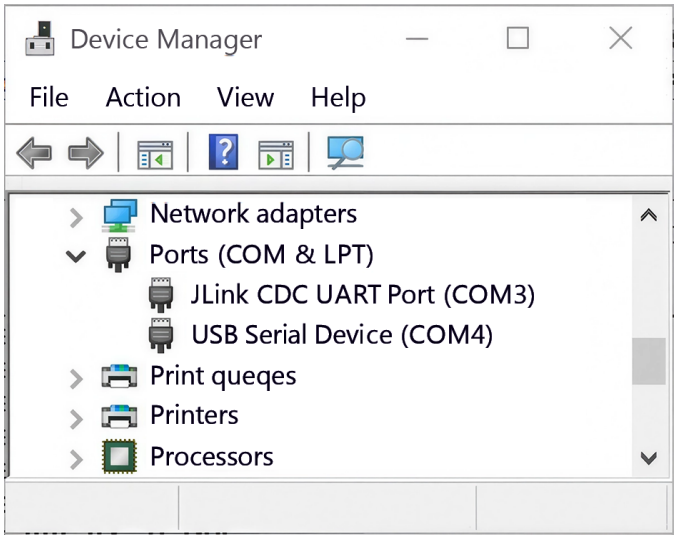


Figure 12.3 : Windows' Device Manager, showing a VCP (COM4)

When using Tera Term, a serial-port must be specified. (Click on: **File** : **New connection**.) The following figure shows a VCP being selected (COM4), in Tera Term.

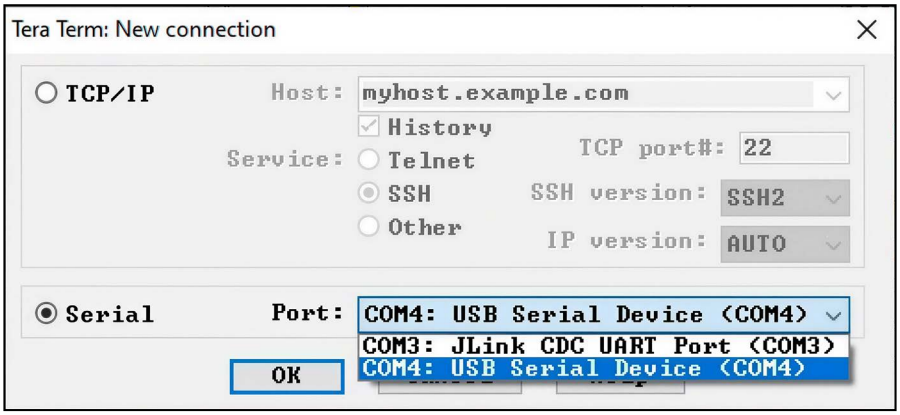


Figure 12.4 : Selecting a VCP, in Tera Term

Now that we understand how messages are sent over USB, let's look at an example, with actual code.

Sending messages using a simple USB API

The chapter's first example-program is `Src/mainVcpSimple.c`. It sends messages to the serial-console on the development-computer. The messages displayed on the serial-console are similar those shown earlier in *Figure 2*. The example-program sends messages over USB by using the API **CDC_Transmit_FS()**. As we'll see, there are limitations to `CDC_Transmit_FS()`. The chapter's second-example shows a better way of sending messages, without those limitations. `CDC_Transmit_FS()` is used here because it's a simple introduction to sending messages over USB. Also, `CDC_Transmit_FS()` illustrates the need for better ways of sending messages over USB, and FreeRTOS can help with that.

Understanding the code

The example-program's `main()` is shown below. It's very similar to the prior example-programs' `main()`. A FreeRTOS task is created, and the scheduler is `started`. The task created is `sendDataTask`, and it sends messages over USB to the serial-console. The only difference with prior `main()` functions is the call to `MX_USB_DEVICE_Init()`, which initializes USB.

```
int main(void)
{
    // Initialize hardware and HAL
    HWInit();

    // USB initialization
    MX_USB_DEVICE_Init();

    // Ensure proper priority-grouping for FreeRTOS
    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

    assert_param(xTaskCreate(sendDataTask, "sendDataTask", STACK_SIZE, NULL,
                             tskIDLE_PRIORITY + 2, NULL) == pdPASS);

    // Start the scheduler - shouldn't return unless there's a problem
    vTaskStartScheduler();

    while(1)
    {
    }
}
```

For the example-program's project, the IDE was used to generate the code needed for using USB. The generated code includes the STM32Cube "USB device library", which is generic for all STM32 MCUs. The code generated for the USB HAL layer is adapted for the MCU being used (STM32F767ZI, in our case). The USB APIs we've seen thus far were generated by the IDE: `MX_USB_DEVICE_Init()` and `CDC_Transmit_FS()`. The latter API was modified by the authors. Modified USB APIs will be noted, but they are treated as black-boxes, so the modifications aren't explained, except in the code-comments.

The task, `sendDataTask`, is shown below.

- First, a spin-loop is run until USB initialization is completed (**lines #10-15**). The spin-loop could have been implemented in `main()`, after calling `MX_USB_DEVICE_Init()`. Instead, the spin-loop is implemented here, to take advantage of FreeRTOS's delay-function.
- Next, in the infinite while-loop (**line #17**), each iteration sends two messages (**lines #26, 35**), then waits for 2 ms (**line #37**). The first message is "message_X: <count>", and the second is "message_Y: <count>". The message count is used to help detect missing messages.
- `unsignedToAscii()` is used in creating the messages (**lines #20, 29**). It converts the message-count to ASCII. It's defined in `Common\Src\unsignedToAscii.c`. The function `sprintf()` to was not used as it can be problematic with FreeRTOS, e.g., it may not be thread-safe.
- The messages are sent using `CDC_Transmit_FS()` (**lines #26,35**). It is in `Drivers/HandsOnRTOS/usbd_cdc_if.c`. Its code-comments specify how to use it.

```
01> void sendDataTask( void* NotUsed)
02> {
03>     #define STRING_BUFFER_SIZE 25
04>     char stringBuffer[STRING_BUFFER_SIZE];
05>     #define NUM_BUFFER_SIZE 11
06>     char numBuffer[NUM_BUFFER_SIZE];
07>     char * numPtr;
08>     . . .
09>
10>     // Ensure USB-initialization is completed.
11>     while(hcdc == NULL)
12>     {
13>         hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
14>         vTaskDelay(10);
```

```
15>     }
16>
17>     while(1)
18>     {
19>         // Create the message to send:  message_X: <messageXcount>
20>         numPtr = unsignedToAscii( ++messageXcount,
21>                                   (numBuffer+NUM_BUFFER_SIZE) );
22>         strcpy(stringBuffer, "message_X: ");
23>         strcpy(stringBuffer+strlen(stringBuffer), numPtr);
24>         strcpy(stringBuffer+strlen(stringBuffer), "\n");
25>
26>         CDC_Transmit_FS((uint8_t*)stringBuffer, strlen(stringBuffer));
27>
28>         // Create the message to send:  message_Y: <messageYcount>
29>         numPtr = unsignedToAscii( ++messageYcount,
30>                                   (numBuffer+NUM_BUFFER_SIZE) );
31>         strcpy(stringBuffer, "message_Y: ");
32>         strcpy(stringBuffer+strlen(stringBuffer), numPtr);
33>         strcpy(stringBuffer+strlen(stringBuffer), "\n");
34>
35>         CDC_Transmit_FS((uint8_t*)stringBuffer, strlen(stringBuffer));
36>
37>         vTaskDelay(2);
38>     }
39> }
```

Running the code

To run the code, first connect the dev-board to the development-computer as shown earlier. Two USB cables are needed. The code can be built and run via Ozone, as in prior example-programs. The build configuration is `vcpSimple`.

After the program starts, the USB connection's VCP will be accessible on the development-computer, as shown earlier. Start Tera Term and connect it to the VCP, as shown earlier.

We'll need to configure Tera Term to recognize the messages' new-line character. To do that, select **Setup** : **Terminal**, and under **New-line** : **Receive**, select **LF**.

The messages scroll by quickly, so to examine them, we'll need to pause the program in Ozone.

The display should look like the figure below.

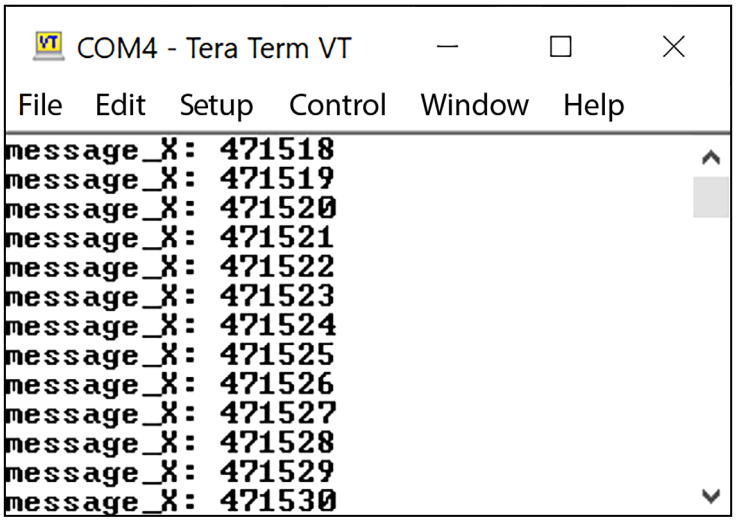


Figure 12.5 : Tera Term, for the example-program

SystemView isn't supported for use with this example-program. If used, incorrect tasks are displayed.

Analyzing the code

From the serial-console (*Figure 12.5*), we can see that the `message_Y` messages are missing. `message_Y` is the second message sent in while-loop iteration.

The cause of the problem can be found by examining `CDC_Transmit_FS()`, which is shown below. We can see that `sendDataTask` should have checked if the value returned from `CDC_Transmit_FS()` is `USBD_BUSY`. In `CDC_Transmit_FS()`, before overwriting the transmit buffer with new data, the function checks to ensure a transfer is not being performed. Data can only be transmitted if a transfer is not already in progress, as indicated by `hcdc->TxState`.

In the task, `message_Y` is sent immediately after `message_X`. So, the `message_Y` messages are not getting transmitted. When `CDC_Transmit_FS()` is called for `message_Y`, the prior `message_X` is still being transmitted.

Also in the task, `vTaskDelay(2)` is called at the end of the while-loop's iteration. The delay reduces the likelihood that a transmit is in progress when `CDC_Transmit_FS()` is called for `message_X`.

```
uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len)
{
    uint8_t result = USBD_OK;
    USBD_CDC_HandleTypeDef *hcdc = NULL;
    hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
    . . .
    // Test if the last transmit is still in progress.
    if (hcdc->TxState != 0){
        return USBD_BUSY;
    }

    // Transmit Buf's contents
    USBD_CDC_SetTxBuffer(&hUsbDeviceFS, Buf, Len);
    result = USBD_CDC_TransmitPacket(&hUsbDeviceFS);

    return result;
}
```

Improving the code

A simple way to fix our code is to use polling. We could simply wrap each and every call to `CDC_Transmit_FS` in a conditional statement to check whether the transmit-process is in-progress :

```
int count = 10;
while(count > 0)
{
    count--;
    if(CDC_Transmit_FS((uint8_t*)"test\n", 5) == USBD_BUSY)
        vTaskDelay(2);
    else
        break;
}
```

This approach is inefficient when attempting to transmit messages back to back, for the following reasons:

- Using no delay would result in continuous polling, which wastes processing-resources and consumes power.
- Increasing `vTaskDelay` tends to decrease the amount of polling.
- However, increasing `vTaskDelay` also tends to increase the amount of unnecessary delay.

The next example-program shows how a driver can be created for sending USB messages, which avoids the use of polling.

Creating a VCP driver, using a stream-buffer

The next example-program is in `Src/mainVcpStreamBuffer.c`. It implements a driver for sending messages over USB, to a VCP on the USB host. The driver uses a stream-buffer to hold messages that are waiting to be sent over USB. The messages are sent by a task that takes messages from the stream-buffer, and transmits them over the USB connection. After the task transmits a message, it waits for a signal that indicates the transmission process is done. The signal is generated by an ISR, which is run via an interrupt from the USB controller. The interrupt is generated when the USB transmission-process is done.

The driver is implemented in `Drivers/HandsOnRTOS/vcpDriver.c`, and the driver is referred to as the `vcpDriver`.

Understanding the app

From our current example-program (`mainVcpStreamBuffer.c`), the function `main()` is shown below.

`mainVcpStreamBuffer.c` is very similar to the prior example-program's `mainVcpSimple.c`. Their `main()` functions are identical, except that `main()` now calls `vcpInit()`, the driver's initialization-function. `vcpInit()` initializes USB, and the driver. In addition, `SEGGER_SYSVIEW_Conf()` is called to initialize SystemView. As in the prior example-program, a task is created (`sendDataTask`), and the FreeRTOS scheduler is started. The task `sendDataTask` sends data over the USB connection.

```
int main(void)
{
    // Initialize the hardware and HAL
    HWInit();
```

```
    // Initialize USB, and the driver vcpDriver.c
    vcpInit();

    SEGGER_SYSVIEW_Conf();

    // Ensure proper priority grouping for FreeRTOS
    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

    assert_param(xTaskCreate(sendDataTask, "sendDataTask", STACK_SIZE, NULL,
                             tskIDLE_PRIORITY + 2, NULL) == pdPASS);

    // Start the scheduler - shouldn't return unless there's a problem
    vTaskStartScheduler();

    while(1)
    {
    }
}
```

The task `sendDataTask` is shown below. In its infinite while-loop (**line #10**), each iteration sends two messages, then delays for 2 ms. This task is identical to the prior example-program's task, except the messages are sent by calling `vcpSend()` (**lines #19, 28**). That function is part of the driver `vcpDriver`. The messages sent are "`message_A: <count>`" and "`message_B: <count>`". The delay is used to slow down the loop, to avoid overflow in SystemView (**line #30**).

```
01> void sendDataTask( void* NotUsed)
02> {
03>     #define STRING_BUFFER_SIZE 25
04>     char stringBuffer[STRING_BUFFER_SIZE];
05>     #define NUM_BUFFER_SIZE 11
06>     char numBuffer[NUM_BUFFER_SIZE];
07>     char * numPtr;
08>     . . .
09>
10>     while(1)
11>     {
12>         // Create the message to send:  message_A: <messageAcount>
```

```
13>         numPtr = unsignedToAscii( ++messageACount,
14>                                   (numBuffer+NUM_BUFFER_SIZE) );
15>         strcpy(stringBuffer, "message_A: ");
16>         strcpy(stringBuffer+strlen(stringBuffer), numPtr);
17>         strcpy(stringBuffer+strlen(stringBuffer), "\n");
18>
19>         vcpSend((uint8_t*) stringBuffer, strlen(stringBuffer));
20>
21>         // Create the message to send:  message_B: <messageBcount>
22>         numPtr = unsignedToAscii( ++messageBCount,
23>                                   (numBuffer+NUM_BUFFER_SIZE) );
24>         strcpy(stringBuffer, "message_B: ");
25>         strcpy(stringBuffer+strlen(stringBuffer), numPtr);
26>         strcpy(stringBuffer+strlen(stringBuffer), "\n");
27>
28>         vcpSend((uint8_t*) stringBuffer, strlen(stringBuffer));
29>
30>         vTaskDelay(2);
31>     }
32> }
```

Understanding the driver

The driver, `vcpDriver`, is implemented in `Drivers/HandsOnRTOS/vcpDriver.c`. The driver is initialized by the function `vcpInit()`, which is shown below. The actions it performs are:

- USB is initialized by calling `MX_USB_DEVICE_Init()`, as in the prior example-program.
- The stream-buffer, `vcpTransmitStreamBuffer`, is created. It will hold the messages that are to be sent. The second argument specifies the trigger-level, which is 1. For blocking receives, at least one byte should be returned (unless timeout occurs). The trigger-level is described in *Chapter 11, More Efficient Drivers and ISRs*.
- A semaphore is created, `vcpTransmitCompleteSemaphore`. It is used to signal when the USB transmit-process completes, after a message is transmitted.
- The task, `vcpTransmitTask`, is created. It removes messages from the stream-buffer, and it transmits those messages over the USB connection. The task is given the highest possible priority, to ensure messages are transmitted as soon as possible.

```
void vcpInit( void )
{
    // Initialize USB
    MX_USB_DEVICE_Init();

    // Create stream-buffer
    vcpTransmitStreamBuffer = xStreamBufferCreate( BUFFER_LEN, 1 );
    assert_param( vcpTransmitStreamBuffer != NULL);

    // Create semaphore for signalling transmit-complete
    vcpTransmitCompleteSemaphore = xSemaphoreCreateBinary();
    assert_param( vcpTransmitCompleteSemaphore != (SemaphoreHandle_t) NULL);

    // Create the task.
    assert_param(xTaskCreate( vcpTransmitTask, "vcpTransmitTask", 256, NULL,
                              (configMAX_PRIORITIES-1),
                              &vcpTransmitTaskHandle) == pdPASS);
}
```

`vcpDriver` also contains the function `vcpSend`, which the user calls to send a message over the USB connection. `vcpSend` is shown below.

- `vcpSend` first checks if the task `vcpTransmitTask` has completed its initialization. If not, a spin-loop is run until initialization is completed. This avoids filling and overflowing the stream-buffer before messages can be sent.
- `vcpSend` then copies the message into the stream-buffer `vcpTransmitStreamBuffer`. The call to `xStreamBufferSend` is not blocking (the parameter `xTicksToWait` is 0). Any bytes from the message that don't fit in the stream-buffer will be dropped.

```
uint32_t vcpSend(uint8_t * Buff, uint16_t Len)
{
    uint32_t numBytesCopied;

    // If needed, spin until vcpTransmitTask has finished its initialization
    while (vcpTransmitTask_initialized == 0){
        vTaskDelay(1);
```

```
    }

    numBytesCopied = (uint32_t) xStreamBufferSend( vcpTransmitStreamBuffer,
                                                   Buff, Len, 0);
    return numBytesCopied;
}
```

The driver defines the call-back function `vcpTransmitComplete` (shown below). When a USB transmit completes, an interrupt is generated, and its ISR will call this call-back function. The function gives the semaphore `vcpTransmitCompleteSemaphore`. The semaphore is used to signal the task `vcpTransmitTask`, to indicate its last transmit has completed.

```
void vcpTransmitComplete( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    // Give the semaphore, to indicate the transmit completed
    xSemaphoreGiveFromISR( vcpTransmitCompleteSemaphore,
                           &xHigherPriorityTaskWoken);

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

The remaining part of `vcpDriver` is its task, `vcpTransmitTask`, which receives messages from the stream-buffer, and transmits them over the USB connection. The first part of the task performs initialization, and it's shown below.

- First, a spin-loop is run until USB initialization is completed. It is implemented here, instead of in `vcpInit()`, to take advantage of FreeRTOS's delay-function.
- The call-back function is set-up, and the USB interrupt-priority.
- Finally, a flag is set to indicate the task has been initialized. The flag is also used in `vcpSend()`.

```
void vcpTransmitTask( void* NotUsed)
{
    uint16_t numBytes;
    USBD_CDC_HandleTypeDef *hcdc = NULL;

    // Ensure USB-initialization is completed.
```

```
    while(hcdc == NULL)
    {
        hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
        vTaskDelay(1);
    }

    // Setup the callback for when USB transmission completes
    hcdc->TxCallBack = vcpTransmitComplete;

    // Ensure the USB interrupt-priority is low enough to allow for
    // FreeRTOS API calls within the ISR
    NVIC_SetPriority(OTG_FS_IRQn, 6);

    // Indicate that the task has been initialized
    vcpTransmitTask_initialized = 1;
```

To facilitate the call-back, the USB device-library had to be modified (`usbd_cdc.c` and `usbd_cdc.h`). It appears that more recent versions of the USB device-library provide a call-back for transmission completion (`CDC_TransmitCplt_FS`), so those versions do not need to be modified.

The second part of the task `vcpTransmitTask` is shown below. It's the task's infinite-loop:

- First, a message is received from the stream-buffer. An unbounded blocking wait is used. Up to `BUFFER_LEN` bytes of data will be copied into `tempBuffer`, when at least 1 byte is available. The latter constraint is the stream-buffer's **trigger-level**, which was specified when the stream-buffer was created.
- The data received from the stream-buffer is then transmitted, using two APIs from the STM32 USB device-library. The first API records the data's address and length. The second API initiates the data-transmit process.
- After initiating the data-transmit process, the task waits on the semaphore that signals completion of the transmit process. In the `xSemaphoreTake()` call, the wait is unbounded. Once the semaphore is taken, the next message can be sent, so the current loop-iteration will complete, and the next iteration will begin.

```
    while(1)
    {
        // Wait forever for data to become available in the stream-buffer
        numBytes = (uint16_t) xStreamBufferReceive( vcpTransmitStreamBuffer,
                                                    tempBuffer, BUFFER_LEN,
```

```
                                                    portMAX_DELAY );

            // Transmit the data
            USBD_CDC_SetTxBuffer(&hUsbDeviceFS, tempBuffer, numBytes);
            USBD_CDC_TransmitPacket(&hUsbDeviceFS);

            // Wait for the semaphore that indicates USB transmission is done.
            xSemaphoreTake(vcpTransmitCompleteSemaphore, portMAX_DELAY);
        }
    }
```

In the task, a semaphore is used to signal the USB transmission is completed. A faster alternative to semaphores is task-notifications, using `xTaskNotifyWait()` and `xTaskNotify()`. However, these task-notification APIs cannot be used in this task, due to the blocking call to `xStreamBufferReceive()`.A limitation of stream-buffers is that a stream-buffer API that puts the calling task into the Blocked state can change the calling task's notification-state and value. The FreeRTOS reference-manual does not specify how the notification-state could be changed. So, in `vcpTransmitTask`, the call to `xStreamBufferReceive()` could change the task's notification-state in an unspecified way. Such a change could potentially conflict with other uses of the task's notification-state, including use by `xTaskNotifyWait()` and `xTaskNotify()`.

Starting in FreeRTOS v10.4.0, tasks can have an array of task-notification values. And, stream-buffers use the task-notification at array index 0. For the current example-program, a task-notification could be used instead of a semaphore, if the array index is not 0. *Chapter 9* has more information on task-notification arrays, including their use with the book's IDE projects.

In the FreeRTOS documentation, the stream-buffer's usage requirements and limitations are specified in the introduction to stream-buffers: `https://www.freertos.org/RTOS-stream-buffer-example.html`.

Running the code

This example-program can be run in the same way as the prior example-program. SystemView can be used. To avoid overflow in SystemView, stop the Ozone debug-session before starting SystemView.

In examining the serial-console, it does not appear that messages are being dropped (as shown below). In the next example-program, performance-data is collected, and tests are run to determine the drop rate.

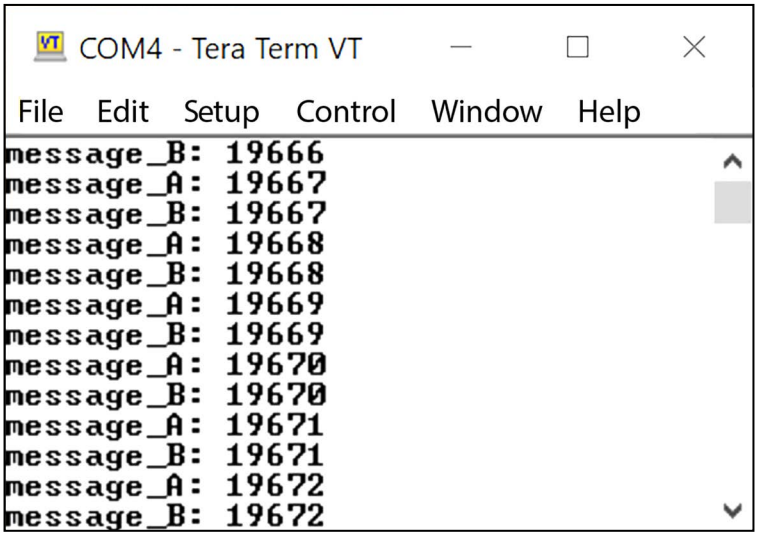


Figure 12.6 : Tera Term display, for the example-program

Analyzing the driver

The driver has two limitations that warrant discussion and improvement:

- The function `vcpSend()` is passed a message to send, and it sends the message to the stream-buffer. However, if the message doesn't fit in the stream-buffer, the excess bytes are simply dropped.
- A limitation of stream-buffers is that they can only be used by one sender at a time (one ISR or one task), and one receiver at a time (one ISR or one task). (This was described in *Chapter 11*.) These requirements are met in the current example-program, as the only task that calls `vcpSend()` is `sendDataTask`. Also, `vcpTransmitTask` is the only task that receives from the stream-buffer. However, by only allowing one task to send messages to the stream-buffer, this driver-design has limited applicability. For many peripheral drivers, it is necessary that multiple tasks be able to send data.

The next example-program improves upon the current example-program, by removing some of these limitations.

Enhancing the VCP driver, for use by multiple tasks

The next example-program is `Src/mainVcpStreamBufferMultiTask.c`. The driver that accompanies it is `Drivers/HandsOnRTOS/vcpDriverMultiTask.c`.

This example-program is very similar to the prior example-program, except it has these added features:

- `vcpSend()` can be called by multiple tasks.
- When `vcpSend()` is called, if the stream-buffer does not have enough space, `vcpSend()` can wait until space is available.
- `vcpSend()` will send all of the input data, or none of it.
- The example-program's `main()` creates two tasks. Both tasks call `vcpSend()`. The two tasks send data over the USB connection, concurrently.

The example-program has a dozen global-variables that are used for testing and performance analysis. For now, we'll ignore them and focus on the multitasking. The global-variables will be discussed later in the two sections on performance analysis.

Understanding the app

As with the prior example-program, this example-program has an app that sends data over USB. The app uses the example-program's USB driver to send the data. The app is in `mainVcpStreamBufferMultiTask.c`.

`mainVcpStreamBufferMultiTask.c` has two functions. `main()` initializes the system, creates two tasks, and starts the FreeRTOS scheduler. The function `sendDataTask()` defines the tasks. Both tasks run that function.

`main()` is shown below. The part that is new, relative to the prior example-program, is that `main()` calls `xTaskCreate()` twice. Both calls specify the task's function as `sendDataTask()`. The task's names are `sendDataTask #1` and `sendDataTask #2`. These names are used in SystemView. Each task needs to know which task it is, i.e., `#1` or `#2`. The task's identifier is specified in `xTaskCreate()`, in its fourth parameter. That parameter is passed to the task, i.e., `(void*)1` and `(void*)2`.

Also, the `vcpInit()` called here is in `vcpDriverMultiTask.c`. It is different than the `vcpInit()` used in the prior example-program.

```
int main(void)
{
    // Initialize the hardware and HAL
    HWInit();

    // Initialize the USB VCP connection, and driver
    vcpInit();

    // Ensure proper priority grouping for FreeRTOS
    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

    assert_param(xTaskCreate(sendDataTask, "sendDataTask #1", STACK_SIZE,
                             (void*)1, tskIDLE_PRIORITY + 2,
                              NULL) == pdPASS);
    assert_param(xTaskCreate(sendDataTask, "sendDataTask #2", STACK_SIZE,
                             (void*)2, tskIDLE_PRIORITY + 2,
                              NULL) == pdPASS);

    // Start the scheduler - shouldn't return unless there's a problem
    vTaskStartScheduler();

    while(1)
    {
    }
}
```

The task `sendDataTask()` is shown below. It creates a message (**lines #28-34**), and it sends the message over USB by calling `vcpSend()` (**lines #36-37**). `unsignedToAscii()` is used in creating the message-text. The message includes the task number and the message-count, e.g., "`task #1: <count>\n`" (**lines #28-34**).

The message-buffer is initialized to hex zeros (**line #29**). This way, the bytes after the message-text will always be hex zeros. With the serial-console Tera Term, any bytes sent which are hex zero will not be displayed.

The message-text is at the beginning of the task's message-buffer, which is 1024 bytes long (**lines #3-6**). Each task has its own message-buffer. The message-buffer is defined as static, and not as local, to avoid consuming stack space. Two message-buffers are needed, so they are defined using a two-dimensional array. The array's first dimension is indexed by the task-number, minus 1. For instance, the message-buffer for `Task #1` is `sendBuffers[0]`.

Each task has its own counter, which records the number of messages sent (**line #7**). The counter is defined as the static array `sendCount[2]`. Using a static array makes the variables easy to access in Ozone, for performance analysis.

`vcpSend()` copies the specified data to the driver's stream-buffer, similar to the prior example-program (**lines #36-37**). The first parameter is a pointer to the data to be sent, and the second parameter specifies the number of bytes to send. The third parameter specifies the maximum wait-time, in ticks. 0 specifies no wait-time, and `portMAX_DELAY` specifies there is for no limit on the maximum-wait-time. For the call to `vcpSend()`, shown here, the first 100 bytes of the task's `sendBuffers[]` will be sent. The maximum wait-time is specified as 100 ticks.

```
01> void sendDataTask( void* Number)
02> {
03>     // Make the send-buffer the same length as the driver's stream-buffer.
04>     #define SEND_BUFFER_LEN  VCP_DRIVER_MULTI_TASK_BUFFER_LEN
05>     // Define two buffers, one for each task
06>     static char sendBuffers[2][SEND_BUFFER_LEN] = {0};
07>     static uint32_t sendCount[2] = {0};
08>
09>     #define NUM_BUFFER_SIZE 11
10>     char numBuffer[NUM_BUFFER_SIZE];
11>     char * numPtr;
12>     . . .
13>     uint32_t taskNumber;
14>     taskNumber = (uint32_t) Number;
15>
16>     char * sendBufferPtr;
17>     sendBufferPtr = sendBuffers[taskNumber-1];
18>
19>     // In the message prefix, set the task number, e.g., "task #1: "
20>     char messagePrefix[] = "task #_: ";
```

```
21>     messagePrefix[6] = '0'+ taskNumber;
22>
23>     while(1)
24>     {
25>         . . .
26>         sendCount[taskNumber-1]++;
27>
28>         // Create the message to send:
29>         memset( sendBufferPtr, 0, VCP_DRIVER_MULTI_TASK_BUFFER_LEN );
30>         numPtr = unsignedToAscii( sendCount[taskNumber-1],
31>                                   (numBuffer+NUM_BUFFER_SIZE) );
32>         strcpy( sendBufferPtr, messagePrefix);
33>         strcpy( (sendBufferPtr+strlen(sendBufferPtr)), numPtr);
34>         strcpy( (sendBufferPtr+strlen(sendBufferPtr)), "\n");
35>         . . .
36>         // Call the driver, to send the data over USB
37>         sendResult = vcpSend((uint8_t*) sendBufferPtr, 100, 100);
38>         . . .
39>     }
40> }
```

Understanding the driver

The driver is in `vcpDriverMultiTask.c`. The driver's functions are described here, with a focus on the features that were added to the prior example-program. The driver's functions are listed below and presented in the sections that follow:

- `vcpInit()`: Initializes the USB connection and the driver.
- `vcpSend()`: Sends data to the driver's stream-buffer. The data will get sent over the USB connection.
- `vcpTransmitTask()`: Receives data from the driver's stream-buffer, and transmits the data over the USB connection.
- `vcpTransmitComplete()`: A call-back function that is called when the last USB transmit completes. It works the same as in the prior example-program.
- `vcpResetTimerCallBack()`: A call-back function for a FreeRTOS one-shot timer. It is used in collecting performance-data.

vcpInit()

`vcpInit()` is shown below. It initializes the USB connection and the driver. It's the same as `vcpInit()` in the prior example-program, and it has these additional features:

- A mutex is created (**lines #13-15**). It is used to serialize sends to the stream-buffer.
- An additional semaphore is created (**lines #17-19**). In `vcpSend()`, if there is not enough space to write to the stream-buffer, this semaphore provides a means to block and wait for space.
- A one-shot timer is created and started (**lines #21-28**). It is used in collecting data for performance analysis.

```
01> void vcpInit( void )
02> {
03>     MX_USB_DEVICE_Init();
04>
05>     vcpTransmitCompleteSemaphore = xSemaphoreCreateBinary();
06>     assert_param( vcpTransmitCompleteSemaphore !=
07>                       (SemaphoreHandle_t) NULL);
08>
09>     vcpTransmitStreamBuffer = xStreamBufferCreate(
10>                                VCP_DRIVER_MULTI_TASK_BUFFER_LEN, 1 );
11>     assert_param( vcpTransmitStreamBuffer != NULL);
12>
13>     vcpStreamBufferSendMutex = xSemaphoreCreateMutex();
14>     assert_param(vcpStreamBufferSendMutex !=
15>                     (SemaphoreHandle_t) NULL);
16>
17>     vcpStreamBufferIsEmptySemaphore = xSemaphoreCreateBinary();
18>     assert_param(vcpStreamBufferIsEmptySemaphore !=
19>                     (SemaphoreHandle_t) NULL);
20>
21>     TimerHandle_t oneShotHandle =
22>       xTimerCreate( "vcpResetTimerCallBack", // Name for timer
23>                   (resetTimeInSeconds*1000)/portTICK_PERIOD_MS, // Period
24>                   pdFALSE,                 // Auto-reload flag
25>                   NULL,                    // Unique ID for timer
26>                   vcpResetTimerCallBack);  // Callback function
```

```
27>     assert_param(oneShotHandle != NULL);
28>     xTimerStart(oneShotHandle, 0);
29>
30>     assert_param( xTaskCreate( vcpTransmitTask, "vcpTransmitTask", 256,
31>                                NULL, (configMAX_PRIORITIES-1),
32>                                &vcpTransmitTaskHandle ) == pdPASS );
33> }
```

vcpTransmitTask()

`vcpTransmitTask()` is shown below. It's the same as `vcpTransmitTask` in the prior example-program, but with two additional features. One feature is setting the variable `vcpTransmitTask_initialized`, to indicate the task's initialization is completed (**lines #17-18**). The variable is used by `vcpSend()`, to know when it can start putting data in stream-buffer.

The other feature is setting the semaphore `vcpStreamBufferIsEmptySemaphore`, which indicates the steam-buffer is empty (**lines #28-33**). The call to `xStreamBufferReceive()` will empty the stream-buffer. The semaphore is then set to 1, if it is not 1 already. The semaphore is used by `vcpSend()`, as will be described in the next section.

As before, `vcpTransmitTask()` has higher priority than the tasks that call `vcpSend()`.

The code for collecting performance-data has been omitted.

```
01> void vcpTransmitTask( void* NotUsed)
02> {
03>     static uint8_t tempBuffer[VCP_DRIVER_MULTI_TASK_BUFFER_LEN];
04>     USBD_CDC_HandleTypeDef *hcdc = NULL;
05>     . . .
06>     // If necessary, wait for USB to finish initialization.
07>     while(hcdc == NULL)
08>     {
09>         hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
10>         vTaskDelay(1);
11>     }
12>     // * Setup callback, which is called when USB transmission is
       //complete
13>     hcdc->TxCallBack = vcpTransmitComplete;
14>     // Allow for FreeRTOS API calls within the ISR
15>     NVIC_SetPriority(OTG_FS_IRQn, 6);
```

```
16>
17>     // Indicate data can now be transmitted
18>     vcpTransmitTask_initialized = 1;
19>     . . .
20>     while(1)
21>     {
22>         // Wait forever for data to become available in the stream-buffer
23>         numBytes = (uint16_t) xStreamBufferReceive(
24>                                     vcpTransmitStreamBuffer, tempBuffer,
25>                                     VCP_DRIVER_MULTI_TASK_BUFFER_LEN,
26>                                     portMAX_DELAY);
27>
28>         // Indicate the stream-buffer receive occurred
29>         returnValue = (int32_t) uxSemaphoreGetCount(
30>                             vcpStreamBufferIsEmptySemaphore);
31>         if (returnValue == 0) {
32>            returnValue = xSemaphoreGive(vcpStreamBufferIsEmptySemaphore);
33>         }
34>
35>         . . .
36>         // Transmit the data
37>         USBD_CDC_SetTxBuffer(&hUsbDeviceFS, tempBuffer, numBytes);
38>         USBD_CDC_TransmitPacket(&hUsbDeviceFS);
39>
40>         // Wait for the semaphore that indicates transmission is done.
41>         xSemaphoreTake(vcpTransmitCompleteSemaphore, portMAX_DELAY);
42>         . . .
43>     }
44> }
```

vcpTransmitComplete()

`vcpTransmitComplete()` is shown below. It's a call-back function that is called when the USB-transmit completes. As in the prior example-program, the function gives a semaphore `vcpTransmitCompleteSemaphore`, to indicate the USB-transmit completed.

```
void vcpTransmitComplete( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    // Give the semaphore, to indicate the transmit completed
    xSemaphoreGiveFromISR(vcpTransmitCompleteSemaphore,
                          &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

vcpSend()

Tasks call `vcpSend()` to put data in the driver's stream-buffer, for the data to be sent over the USB connection. In the prior example-program, `vcpSend()` also did this, but it could only be called by one task. The present `vcpSend()` has been redesigned, so that it can be called by multiple tasks.

`vcpSend()` is longer and more complex than most functions in the book's example-programs. `vcpSend()` is presented in four parts, in the next four sections, respectively. In the code listings, the code related to performance-analysis and testing has been omitted. The program file has additional comments, which are not shown here.

vcpSend(): initial processing

The first part of `vcpSend()` is shown below. (It is the first part, of 4 parts.) The code is explained in the sections that follow it.

```
1-01> int32_t vcpSend(uint8_t * Buff, uint16_t Len, TickType_t ticksToWait)
1-02> {
1-03>     . . .
1-04>     // Spin if the task vcpTransmitTask is not yet ready to transmit
1-05>     while (vcpTransmitTask_initialized == 0){
1-06>         vTaskDelay(1);
1-07>     }
1-08>
1-09>     startTickCount = xTaskGetTickCount();
1-10>
1-11>     // Get the mutex, to serialize sends to the stream-buffer
1-12>     if ( xSemaphoreTake(vcpStreamBufferSendMutex, ticksToWait)
1-13>              == pdPASS )
1-14>     {
```

The parameters and return value

The three parameters for `vcpSend()` were described earlier, and they are shown in **line #1**. The first parameter (`Buff`) is a pointer to the data to be sent, and the second parameter (`Len`) specifies the number of bytes to send. The current example-program added the third parameter: `ticksToWait`. It specifies the maximum number of ticks to wait when calling blocking APIs. `vcpSend()` calls two blocking APIs, and the maximum wait-time applies to both of them, collectively, as will be explained.

Also, the input data will only be sent to the stream-buffer if all of the data will fit. The data put in the stream-buffer is contiguous, and data from other tasks is not mixed within it.

The return value from `vcpSend()` is a signed integer. Negative values indicate the send failed, and the specific value specifies the cause. If the send worked, the return-value is the number of bytes sent, i.e., the value in the parameter `Len`.

Initialization

The task `vcpTransmitTask` receives data from the driver's stream-buffer, and the task transmits the data over the USB connection. The task's initialization process does not occur immediately. When the task's initialization is complete, it sets the global variable `vcpTransmitTask_initialized`, to 1.

When `vcpSend()` starts, it will spin if `vcpTransmitTask` is not yet initialized (**lines #4-7**). Waiting for initialization to complete guards against filling the stream-buffer during initialization, and then dropping data.

Using a mutex to serialize calls to xStreamBufferSend()

In `vcpSend()`, bytes are sent to the stream-buffer by calling `xStreamBufferSend()`. For that API to be used by multiple tasks, there are two implementation requirements:

- The calls to `xStreamBufferSend()` must be in a critical section, so the calls are only made by one task at a time.
- The calls to `xStreamBufferSend()` must be non-blocking, i.e., specify a maximum wait-time of 0.

In `vcpSend()`, the former requirement is implemented by using a mutex (`vcpStreamBufferSendMutex`). The latter requirement is implemented by using a semaphore. The semaphore makes it possible for `vcpSend()` to wait for space to be available, if necessary. The semaphore-use is explained later.

As mentioned earlier, in the FreeRTOS documentation, the stream-buffer's usage requirements and limitations are specified in the introduction to stream-buffers: `https://www.freertos.org/RTOS-stream-buffer-example.html`.

The mutex `vcpStreamBufferSendMutex` is taken in **lines #11-13**. The related critical-section starts at **line #14**.

vcpSend(): implementing a maximum wait-time

The next part of `vcpSend()` is shown below. (It is the second part, of 4 parts.)

In `vcpSend()`, the function's maximum wait-time is specified by the parameter `ticksToWait`. It includes the time spent waiting for the mutex, and the time spent waiting for space to be available in the stream-buffer.

The code below calculates the time spent waiting for the mutex, and it calculates the time that is available for waiting for stream-buffer space. An explanation follows.

```
2-01>         // Calculate how long xSemaphoreTake() took
2-02>         if ( (ticksToWait != 0) && (ticksToWait != portMAX_DELAY) ){
2-03>             currentTickCount = xTaskGetTickCount();
2-04>
2-05>             if (currentTickCount >= startTickCount) {
2-06>                 elapsedTickCount = currentTickCount - startTickCount;
2-07>             }
2-08>             // Else, the tick-counter wrapped
2-09>             else {
2-10>                 elapsedTickCount = (portMAX_DELAY - startTickCount)
2-11>                                    + currentTickCount + 1;
2-12>             }
2-13>         }
2-14>
2-15>         // Determine remaining waiting-time
2-16>         if ( (ticksToWait == 0) ||
2-17>              (ticksToWait == portMAX_DELAY) ) {
2-18>             remainingWaitingTime = ticksToWait;
2-19>         }
2-20>         else if (elapsedTickCount < ticksToWait) {
2-21>             remainingWaitingTime = ticksToWait - elapsedTickCount;
2-22>         }
2-23>         else {
2-24>             remainingWaitingTime = 0;
2-25>         }
```

The parameter `ticksToWait` is of type `TickType_t`. Many FreeRTOS APIs have a parameter for specifying the maximum wait-time (i.e., the maximum time in the blocked state). The parameter's type is `TickType_t`. It is an unsigned int, and in our system, it is 32 bits.

The parameter `ticksToWait` can have three types of values. (We'll refer to them as the *parameter-value types.*):

- 0 : specifies to not block (no wait-time)
- `portMAX_DELAY` : specifies there is for no limit on the wait time. (It is the largest integer that can be stored with `TickType_t`.)
- *tick-count* : any value other than 0 or `portMAX_DELAY` (0 < tick-count < `portMAX_DELAY`). It specifies the maximum wait-time, in tick-counts.

The maximum wait-time is implemented differently for each parameter-value type. The implementation for the tick-count is explained first. The code is described, and the line numbers are cited.

If `ticksToWait` specifies a tick-count, then the time spent waiting for the mutex must be calculated. The calculation is made in **lines #1-13**.

To measure how much time was spent, the FreeRTOS tick-counter is used. It is a system variable that is incremented at each FreeRTOS tick. The counter wraps to 0 after reaching its maximum value. The maximum value is defined in `portMAX_DELAY` (it is in FreeRTOS's `portmacro.h`).

In calculating the time spent waiting for the mutex, **lines #8-12** handle the case of a wrapped tick-counter.

The wait-time calculated for the mutex is stored in `elapsedTickCount` (**lines #6, 10**). Note the calculation is an approximation of the amount of time the API-call is blocked, waiting for the mutex. The time between `currentTickCount` and `startTickCount` can include other causes, such as an ISR or blocking for a higher priority task. Also, the calculated time's precision is a tick-count, so the calculated time can be almost 1 tick-count more than the actual time.

Having calculated the `elapsedTickCount`, we now need to calculate the remaining wait-time, which can be used to wait for stream-buffer space. The calculation is in **lines #15-25**, and the result is recorded in `remainingWaitingTime`.

If the parameter `ticksToWait` is 0 or `portMAX_DELAY`, then `remainingWaitingTime` is simply assigned that value (**lines #16-19**). If `ticksToWait` has a tick-count, then `remainingWaitingTime` is calculated as `ticksToWait` minus the time spent in the API-call for the mutex, i.e., `elapsedTickCount` (**lines #20-22**). Provision is made for the possibility of `elapsedTickCount` being more than `ticksToWait` (**lines #23-25**). In that case, remainingWaitingTime is set to 0.

vcpSend(): using a semaphore to wait for stream-buffer space

The next part of `vcpSend()` is shown below. (It is the third part, of 4 parts.)

In `vcpSend()`, the stream-buffer may not have enough space for the input-data. In that case, a semaphore is used to wait for adequate space. This section describes how the semaphore is used.

The semaphore is `vcpStreamBufferIsEmptySemaphore`, and it indicates the stream-buffer has been emptied. Whenever `vcpTransmitTask()` receives from the stream-buffer, the stream-buffer is emptied. Also, `vcpTransmitTask()` sets the semaphore to 1, if it's not 1 already. (This was shown earlier.)

```
3-01>         // * Set the semaphore vcpStreamBufferIsEmptySemaphore:
3-02>         //   * Set it to 0, if it's not 0 already.
3-03>         returnValue = (int32_t) uxSemaphoreGetCount(
3-04>                                     vcpStreamBufferIsEmptySemaphore);
3-05>         if (returnValue == 1) {
3-06>             // Semaphore is 1, so call xSemaphoreTake() to set it to 0
3-07>             returnValue = (int32_t) xSemaphoreTake(
3-08>                                   vcpStreamBufferIsEmptySemaphore, 0 );
3-09>             . . .
3-10>         }
3-11>
3-12>         bytesAvailable = xStreamBufferSpacesAvailable(
3-13>                               vcpTransmitStreamBuffer);
3-14>
3-15>         // If there is not enough space in the stream-buffer,
3-16>         // and there is remaining waiting-time,
3-17>         // then wait for the stream-buffer to be emptied.
3-18>         if ((bytesAvailable < Len) && (remainingWaitingTime > 0)) {
3-19>             . . .
3-20>             // * Take the semaphore that indicates vcpTransmitTask()
3-21>             //   has emptied the stream-buffer.
3-22>             // * Set the maximum wait-time to remainingWaitingTime
3-23>             if( xSemaphoreTake(vcpStreamBufferIsEmptySemaphore,
3-24>                                remainingWaitingTime) == pdPASS ){
3-25>                 . . .
3-26>                 // The whole stream-buffer is available
3-27>                 bytesAvailable = rxBuffLen;
```

```
3-28>                 . . .
3-29>             }
3-30>         }
3-31>
```

Lines #1-10 involve setting the semaphore `vcpStreamBufferIsEmptySemaphore` to 0, if it's not zero already. This allows us to know that the semaphore was 0 at some point during **lines #1-10**. After that point, if the semaphore becomes 1, we know the stream-buffer was emptied.

At **lines #12-13**, `xStreamBufferSpacesAvailable()` is called to determine how many bytes are available in the stream-buffer.

In `vcpSend()`, determining the stream-buffer's available space is challenging due to the possibility of the stream-buffer being emptied at almost any point while `vcpSend()` is running. One reason this can occur is that `vcpTransmitTask()` has a higher priority than the tasks that run `vcpSend()`.

In **lines #15-30**, `vcpSend()` checks if the stream-buffer has enough space for the input-data. If it does not, and if there is remaining wait-time, then `vcpSend()` will wait for the stream-buffer to be emptied.

At **line #18**, the condition `(bytesAvailable < Len)` will be true if the stream-buffer doesn't have enough space for the input-data. In addition, the condition `(remainingWaitingTime > 0)` is true if there is remaining waiting time. When both conditions are true, `xSemaphoreTake()` will be called for the semaphore (**lines #23-24**). `xSemaphoreTake()` is used to wait for the stream-buffer to be emptied.

At **lines #23-24**, the call to `xSemaphoreTake()` is a blocking call, and the ticks-to-wait is specified as `remainingWaitingTime`. The API-call will block up to that many ticks, waiting for the stream-buffer to be emptied. `vcpTransmitTask()` has higher priority than the tasks that call `vcpSend()`. So, if `xSemaphoreTake()` is blocked, and `vcpTransmitTask()`receives from the stream-buffer, then the stream-buffer will be emptied. Also, `vcpTransmitTask()` will set the semaphore to 1, and `xSemaphoreTake()` will be successful.

Also at **line #18**, if the condition is true, then it is known that the stream-buffer did not have enough space at some point during **line #12**. After that point, the stream-buffer could potentially be emptied, up to when `xSemaphoreTake()` is run, on **lines #23-24**. If the stream-buffer is emptied during that window (during **line #12** to **lines #23-24**), then the `xSemaphoreTake()` will be successful. Also, since the stream-buffer is empty, `bytesAvailable` is set to the length of the stream-buffer (**line #27**).

At the end of this code, at **line #31**, if `(bytesAvailable >= Len)`, then we know there is enough room in the stream-buffer, for the input-data. The condition could be true due to the value assigned to `bytesAvailable` at **line #12**, or at **line #27** after waiting for space to become available.

The driver has several features that restrict how the stream-buffer can change while `vcpSend()` is running. The driver's semaphore-use relies on these restrictions. While `vcpSend()` is running, the following properties hold:

- While `vcpSend()` holds the mutex `vcpStreamBufferSendMutex`, no other task will add to the stream-buffer.
- `vcpTransmitTask()` is the only task that can receive from the stream-buffer.
- If `vcpTransmitTask()` receives from the stream-buffer, the stream-buffer will be emptied.
- So, in the `vcpSend()` critical-section, up to when `xSemaphoreTake()` is called on **lines #23-24**, the stream-buffer's contents can only stay the same, or be emptied.

This concludes our description of how a semaphore is used to synchronize the stream-buffer use, by `vcpSend()` and `vcpTransmitTask()`. There's one remaining feature to describe, for the driver.

vcpSend(): sending to the stream-buffer

The last part of `vcpSend()` is shown below. (It is the fourth part, of 4 parts.)

If the stream-buffer has enough space for the input-data, then `xStreamBufferSend()` is called to send the data (**lines #1-6**).

The critical-section needed for calling `xStreamBufferSend()` is ended at **line #11**, by giving the mutex.

If the data can't be sent to the stream-buffer, error codes are returned (**lines #8, 14**). If data was sent to the stream-buffer, the number of bytes sent is returned (**lines #3-4**). Since all of the input-data was sent, the value returned is the same as the `vcpSend()` parameter `Len`.

```
4-01>         // If there is enough space in the stream-buffer, send the data.
4-02>         if (bytesAvailable >= Len){
4-03>             returnValue = (int32_t) xStreamBufferSend(
4-04>                                 vcpTransmitStreamBuffer, Buff, Len, 0 );
4-05>             . . .
4-06>         }
4-07>         else {
4-08>             returnValue = VCP_SEND_CANNOT_SEND;
4-09>         }
```

```
4-10>
4-11>         xSemaphoreGive(vcpStreamBufferSendMutex);
4-12>     }
4-13>     else{
4-14>         returnValue = VCP_SEND_MUTEX_NOT_AVAILABLE;
4-15>     }
4-16>     return returnValue;
4-17> }
```

Now that we understand the driver, let's look at how the example-program collects performance-data.

Understanding how the performance-data is collected

The example-program collects performance-data, for use in understanding the driver's effectiveness and for use in understanding how to use the driver effectively. The data is analysed to determine how much of the sent-data is dropped, relative to the maximum wait-time specified in `vcpSend()`.

Performance-data is collected in the tasks `sendDataTask` and `vcpTransmitTask`, and in `vcpSend()`. Highlights of the collected data and the related code are presented here. Tests were run to collect performance-data, and the performance analysis is presented later in this chapter.

`sendDataTask` is shown below. Since two tasks run this function, the performance-data is stored in 2-dimensional arrays (**lines #4-8**). There is an array-element for each task. For the performance-data, most of the variables' names start with "test_". Also, the data is declared as static, which makes it easier to find in Ozone (it's in the **Global Data** window).

Performance-data is collected after each call to `vcpSend()`, e.g., counters are incremented (**lines #38-44**). For brevity, not all of the data-collection code is shown here.

If the performance-data is collected when the system starts, the data will be skewed by start-up conditions, such as starting the serial console. To avoid this, the performance-data is reset 30 seconds after the system starts (**lines #17-25**).(For example, counters are reset to zero.) The reset is signalled by a one-shot timer, whose call-back function sets the global variable `vcpResetPerformanceData`, to 1.

In `sendDataTask`, after calling `vcpSend()` and collecting the performance-data, `vTaskDelay()` is called **(lines #46-52)**. A one-tick delay is used for `Task #1`, and two ticks for `Task #2`. Without these delays, the two tasks will send data constantly, and simulate an extreme use of USB with VCP. The delays are used to simulate a less extreme use of USB.

Performance-data is collected in a similar manner, in the task `vcpTransmitTask` and in the function `vcpSend()`. That performance-data is also reset 30 seconds after the system starts.

```
01> void sendDataTask( void* Number)
02> {
03>     . . .
04>     static uint32_t sendCount[2] = {0};
05>     . . .
06>     // Performance-analysis data
07>     static uint32_t test_sendDataTask_sendError[2];
08>     static uint32_t test_sendDataTask_mutexError[2];
09>     . . .
10>     bool resetPerformed = false;
11>     . . .
12>     taskNumber = (uint32_t) Number;
13>     . . .
14>
15>     while(1)
16>     {
17>         // Reset the performance-analysis data
18>         if ( (resetPerformed == false) &&
19>              (vcpResetPerformanceData == 1) )  {
20>             sendCount[taskNumber-1] = 0;
21>             test_sendDataTask_sendError[taskNumber-1] = 0;
22>             test_sendDataTask_mutexError[taskNumber-1] = 0;
23>             . . .
24>             resetPerformed = true;
25>         }
26>         sendCount[taskNumber-1]++;
27>
28>         // Create the message to send:
29>         memset( sendBufferPtr, 0, VCP_DRIVER_MULTI_TASK_BUFFER_LEN );
30>         numPtr = unsignedToAscii( sendCount[taskNumber-1],
```

```
31>                                     (numBuffer+NUM_BUFFER_SIZE) );
32>         strcpy( sendBufferPtr, messagePrefix);
33>         strcpy( (sendBufferPtr+strlen(sendBufferPtr)), numPtr);
34>         strcpy( (sendBufferPtr+strlen(sendBufferPtr)), "\n");
35>         . . .
36>         sendResult = vcpSend((uint8_t*) sendBufferPtr, 100, 0);
37>         . . .
38>         // Record errors returned from vcpSend
39>         if (sendResult == VCP_SEND_CANNOT_SEND){
40>             test_sendDataTask_sendError[taskNumber-1]++;
41>         }
42>         else if (sendResult == VCP_SEND_MUTEX_NOT_AVAILABLE){
43>             test_sendDataTask_mutexError[taskNumber-1]++;
44>         }
45>
46>         // Delays, to avoid back-to-back sends.
47>         if (taskNumber == 1){
48>             vTaskDelay(1);
49>         }
50>         else{
51>             vTaskDelay(2);
52>         }
53>     }
54> }
```

Running the code

This example-program can be run in the same way as the book's prior example-programs. SystemView is not used. In Ozone, when opening the *.jdebug file, there may be a red console-message, "**Warning (98): The symbol location decoder encountered an unsupported operand: 0xFA.**" It appears that this message can be ignored.

When the program starts the green user-LED is turned-on. After 30 seconds, the performance-data will be reset, and the blue user-LED is turned-on. To collect steady-state performance-data, the serial-console should be connected to the VCP, and configured, within about 20 seconds.

The performance-data can be viewed in Ozone. After pausing the firmware, the data can be viewed in the **Global Data** window.

The serial-console's display should look like the figure below.

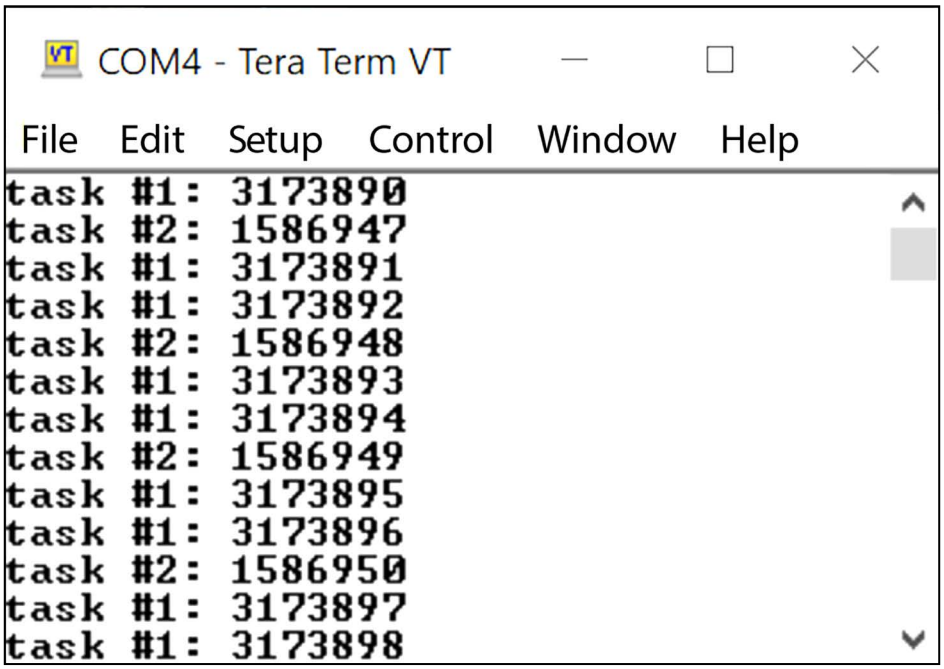


Figure 12.7 : Tera Term display, for the example-program

Analyzing the driver's performance

12 tests were run, and the performance-data is shown in the table below. Following the table, the overall test results are described. Then, the table's columns are described.

Calls to vcpSend						**vcpTransmitTask**	
Arguments		**Delay**	**Send Results**			**xStreamBufferReceive**	
Max Wait-Time	**Bytes Sent**	**After Call**	**OK %**	**Mutex Err %**	**SB Err %**	**Average**	**Max**
0	100	0	28.46%	46.24%	25.30%	348	1,000
0	100	T1:1, T2:2	66.66%	33.33%	0.01%	100	1,000
0	1,024	0	3.43%	43.52%	53.05%	1,024	1,024
0	1,024	T1:1, T2:2	50.98%	0.18%	48.84%	1,024	1,024
100	100	0	99.99%	0.01%	0.00%	1,000	1,000
100	100	T1:1, T2:2	100.00%	0.00%	0.00%	100	1,000
100	1,024	0	99.53%	0.46%	0.01%	1,024	1,024
100	1,024	T1:1, T2:2	100.00%	0.00%	0.00%	1,024	1,024
portMAX_DELAY	100	0	100.00%	0.00%	0.00%	1,000	1,000
portMAX_DELAY	100	T1:1, T2:2	100.00%	0.00%	0.00%	100	1,000
portMAX_DELAY	1,024	0	100.00%	0.00%	0.00%	1,024	1,024
portMAX_DELAY	1,024	T1:1, T2:2	100.00%	0.00%	0.00%	1,024	1,024

Figure 12.8 : Driver performance data

Each test runs the example-program, and each test calls `vcpSend()` in a different way. What we found is is that calling `vcpSend()` with a maximum wait-time of 0 is not viable, as the data drop-rate is too high. (Of the data that is sent, 33% to 97% of it is dropped.) With a maximum wait-time of `portMAX_DELAY`, none of the sent-data is dropped. And, with a maximum wait-time of 100 ticks, the drop-rate was less than 0.5%.

The drop-rates are calculated as the percentage of calls to `vcpSend()` for which the data was not sent to the stream-buffer. That occurs due to time-out. When time-out does not occur, the data is transmitted over USB, and we assume it arrives on the host. When `vcpSend()` is called, it will either send all of the input data to the stream-buffer, or none. In a particular test, all of the calls to `vcpSend()` send the same amount of data, so the drop-rate also represents the percentage of bytes that were not sent to the stream-buffer.

The test-configurations are specified in the table's first three columns. These columns specify how `vcpSend()` was called: a) the maximum wait-time (0, 100, `portMAX_DELAY`), b) the bytes sent (100, 1024), and c) the delay used after the call to `vcpSend()`. The tests used either no delay, or the delays described earlier (i.e., a 1-tick delay in Task #1, and a 2-tick delay in Task #2).

The performance-data is shown in the remaining columns. Under "Send Results", the column "OK %" shows the percentage of `vcpSend()` calls for which the data was sent successfully. The column "Mutex Err %" shows the percentage that failed due to not getting the mutex. The column "SB Err%" shows the percentage that failed due to not getting space in the stream-buffer. The data in these columns can be used to understand the system's capabilities. It can also be used for investigating ways to reduce the drop-rate, e.g., by increasing the maximum wait-time, or by increasing the stream-buffer size.

The columns under "vcpTransmitTask" have performance-data from that task. The performance-data records the amount of data received from the stream buffer, in the call to `xStreamBufferReceive()`. The stream-buffer is 1024 bytes, so it's the most that can be received. For tests in which `vcpSend()` is used to send 100 bytes, the performance-data shows the stream-buffer can be useful in queuing data from multiple calls to `vcpSend()`.

The performance of the USB transmits depends on the USB-host's ability to receive data. The VCP presumably has a buffer for incoming data, and the buffer-size affects the rate at which data can be received. The USB-host's ability to receive data also depends on the speed at which the serial-console reads from the VCP. So, the overall-system's end-to-end performance depends on both: a) the firmware's ability to send data, and b) the USB-host's ability to receive data.

The example-program also collects performance-data for how long the calls to `vcpSend()` take, and how long the USB-transmits take. The time is measured in ticks. This code isn't presented here in the book, but it is commented in the source code. As an example of the timing performance, for each test, it records the maximum duration of a call to `vcpSend()`. Among the tests with a maximum wait-time of `100 ticks`, the maximum duration of the calls to `vcpSend()` ranged from 48 ticks, up to 101 ticks.

The present section's performance-analysis shows some of the ways that performance-data can be collected and analyzed. And, it shows how the performance can vary, based on: a) how the driver is configured (e.g., buffer sizes), b) how `vcpSend()` is used (e.g., bytes sent, max wait-time, and frequency of the calls), and c) the USB-host's ability to receive data. Further testing and analysis could be performed to better understand the driver's capabilities, and to find better ways to configure and use the driver.

Most importantly, the performance-analysis shows that a maximum wait-time of 0 results in an unacceptably high drop-rate. FreeRTOS requires that if multiple tasks send to the stream-buffer, then a maximum wait-time of 0 must be specified for the sends. So, in our USB driver, another mechanism is needed to wait for space to become available in the stream-buffer. The semaphore that we used is one way to do that.

Summary

This chapter showed how FreeRTOS features can be useful in building a driver that is shared by multiple-tasks. A stream-buffer was used to store messages that are to be transmitted over USB. And, a task was implemented to receive messages from the stream-buffer, and transmit them on a USB connection. Once a message is transmitted, the task cannot transmit another message until the prior message is delivered. The task waits for the prior message to be delivered, which is signalled by a FreeRTOS semaphore.

There are two limitations in using a stream-buffer, when multiple tasks send to the stream-buffer. First, the tasks that send to the stream-buffer must be serialized, so that only one task can send at a time. This was accomplished by using a FreeRTOS mutex. The second limitation is that the API used to send to the stream-buffer must be non-blocking. The API used was `xStreamBufferSend()`. When `xStreamBufferSend()` is called, for it to be non-blocking, the maximum wait-time must be 0. Our tests and performance-analysis show that when a maximum wait-time of 0 is used, a high percentage of the calls to `xStreamBufferSend()` fail.

To ensure messages are sent more reliably, a semaphore was used, to be able to wait for space in the stream-buffer, when necessary. The semaphore is used by `vcpSend()` to wait for space, and by `vcpTransmitTask()` to signal when the stream-buffer is emptied. A limitation of this solution is how `vcpSend()` implements a maximum wait-time. When `xSemaphoreTake()` is called for the mutex, the API's execution time is measured, and it is used as an approximation for the API's block-time.

The chapter introduced how the MCU's USB-peripheral can be used to send data. The next section has links to additional information on using USB.

Further reading

The STM32 USB peripheral is complex. Using it involves learning about the USB protocol and architecture, the MCU's USB-peripheral, and the USB libraries. Comprehensive presentations of these topics can be overwhelming. An alternative is to start with simpler hands-on tutorials that focus on specific USB uses and components. For example, the chapter's example-programs use the USB Communication Device Class (CDC), and its PSTN Device Subclass. STM32CubeIDE can generate the code and USB-libraries for using it, and there are on-line tutorials that show how to do that.

Sources on USB from ST are listed in *Appendix B*, *Reference Information*. They include tutorials, an on-line course, example programs, user manuals, and reference manuals.

13

Creating Loose Coupling with Queues

In this chapter, we'll explore how queues can provide natural interface definitions for data exchange between tasks.

We'll create a system that uses the dev-board and development computer, connected by USB. An app on the development computer sends commands to our firmware on the dev-board. The commands specify how to set the brightness for the user LEDs. The app on the development computer is a GUI, and it allows the user to set the brightness for each LED.

In the prior chapter, we created a driver for sending data over USB, to the development computer. In this chapter we'll use the driver to receive data over USB, from the development computer. The driver puts the received data in a stream-buffer. The received data is a series of commands for setting the brightness of the dev-board's user LEDs. A task is implemented to parse the stream-buffer to extract the commands. Each extracted command is sent to a queue. Another task receives commands from the queue and processes them. To process a command, functions are called to set the LEDs' brightness, as specified by the command.

The overall objective for the chapter is to show how FreeRTOS can be used to solve real-life problems, using FreeRTOS primitives. Although the system performs a very simple operation (setting LEDs), the design is extensible to other systems that involve receiving a serial data stream, parsing it, and processing its elements, such as commands.

We will cover the following topics in this chapter:

- Creating a command queue
- Reusing a queue definition for a new target
- Understanding queues as interfaces

The chapter's example-program is in the IDE-project `Chapter_13`. Two USB cables are needed. They are used to connect the dev-board to the development computer, as in the prior chapter.

Creating a command queue

There are two parts to the example-program: an app that runs on the development computer, and firmware that runs on the dev-board. The app is shown in *Figure 13.1*. The app sends commands over USB to the firmware. Each command specifies how the dev-board's user-LEDs are to be displayed (green, red, and blue). For each LED, its brightness is specified. On the slider-bars, 0% is off, 100% is fully on, and partial brightness is in-between (e.g., 50% is half brightness). The slider-bars' index is shown, and it ranges from 0 to 255. The commands sent by the app also specify whether the LEDs should shine continuously or blink.

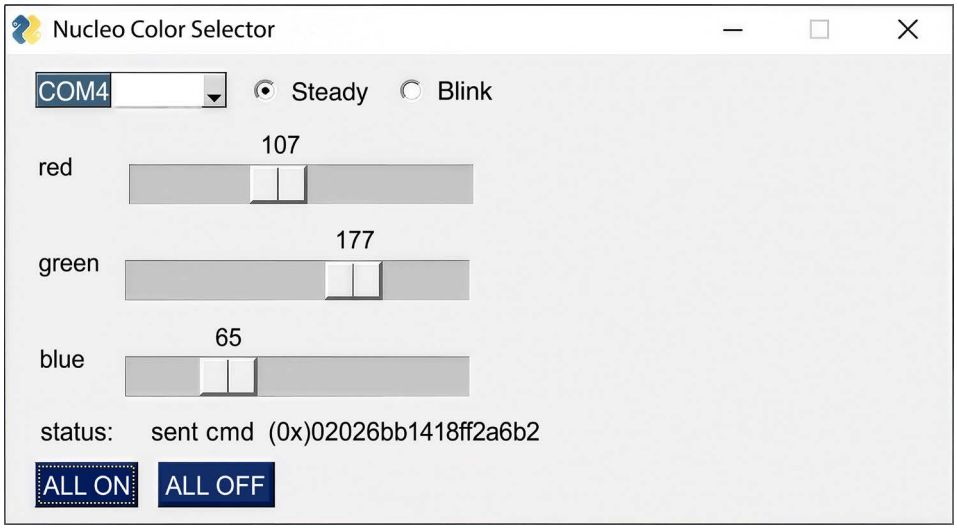


Figure 13.1: The app that sends LED commands to the dev-board

While the example-program itself is very simple, the concepts presented scale extremely well. So, regardless of whether there are only a few commands or hundreds, the same approach can be used to keep an architecture flexible.

The example-program also provides an example of how to keep higher-level code loosely coupled to the underlying hardware. To set the LEDs' brightness, the example-program uses two components. One component processes LED commands. The other component sets the LED hardware, for the specified brightness. Setting the LED hardware involves the use of timers and **Pulse Width Modulation (PWM)**. The component that sets the LED hardware consists of three C-functions, one for each LED (red, green and blue). The three functions' interfaces are defined in a header file. To port the example-program to work with different LED hardware, the command-processing code would not need to be changed, nor the interfaces to the functions that set the LED hardware. The only change needed is the implementation of those three functions. (How the functions work is shown in the next section, "The LED's brightness-functions".)

The example-program's architecture consists of the following major components:

- **A USB driver**: This is the same USB driver that was used in *Chapter 12*. There, the USB driver's feature for sending data was used. The driver also has a feature for receiving data into a stream-buffer, and we'll use that feature here. The driver is at: `Drivers/HandsOnRTOS/vcpDriverMultiTask.c/h`.
- **iPWM**: This is the interface-definition for the three functions that set the LEDs' brightness. It's in `Inc/iPWM.h`.
- **PWM implementation**: These are the functions that set the LEDs' brightness. The functions use timers and PWM to power the LEDs. How PWM works isn't described, and the program's functions are treated as black-boxes. (More information on PWM can be found at the end of the chapter, in the section *Further Reading*.) The functions are in `Src/pwmImplementation.c` and `Inc/pwmImplementation.h`.
- **An LED command-executor**: This program runs as a task. It gets commands from a queue, and it calls the functions that set the LEDs' brightness. A simple state-machine is used to drive the LED states. It uses pointers to the functions for each LED. The program is in `Src/ledCmdExecutor.c`.
- **main()**: This is the `main()` function that ties all of the queues, drivers, and interfaces together, and kicks off the FreeRTOS scheduler. It's in `Src/mainColorSelector.c`.

We'll get into the details of exactly how all of these parts fit together and the details of their responsibilities. But first, let's discuss how the LEDs' brightness is set.

The LEDs' brightness functions

The function that sets the blue LED's brightness is described here. The other LED's brightness-functions work the same way.

The blue LED's brightness-function is `SetBlueDuty()`, and it is shown below. It's in `pwmImplementation.c`. The parameter `DutyCycle` specifies the LED's brightness, on a scale of 0.0 to 100.0. The function powers the LED by using Timer 4 and PWM.

```
void SetBlueDuty( float DutyCycle )
{
    TIM4->CCR2 = DutyCycle/100.0 * 65535;
}
```

The LEDs' brightness-functions are called via function pointers. For each LED brightness-function, its function-pointer is stored in a struct of type `iPWM`, as shown below. (It's in `iPWM.h`.)

```
typedef void (*iPwmDutyCycleFunc)( float DutyCycle );

typedef struct
{
   const iPwmDutyCycleFunc SetDutyCycle;
}iPWM;
```

One `iPWM struct` is declared for each LED brightness-function. The `iPWM struct` for `SetBlueDuty()` is shown below. (It's in `pwmImplementation.c`.)

```
iPWM BluePWM = {.SetDutyCycle = SetBlueDuty};
```

Using the `struct BluePWM`, the function `SetBlueDuty()` could be called like this:

```
BluePWM.SetDutyCycle(100);
```

Currently, the struct `iPWM` only consists of a single function-pointer, of type `iPwmDutyCycleFunc`. And, the pointer is defined as a constant. So, once the `iPWM struct` is initialized, the pointer can never be changed. This helps guarantee the pointer won't be overwritten, and it won't ever be necessary to check the pointer isn't NULL.

Wrapping the function-pointer in a `struct` such as `iPWM` provides the flexibility of adding additional functions for setting the LED's brightness. We'll be able to pass a single pointer to the `iPWM` struct to functions, rather than individual function-pointers.



If you are creating an **interface** definition that will be shared with other developers, it is important to be very careful to coordinate and communicate changes with your team!

In `SetBlueDuty()`, the `DutyCycle` parameter is defined as `float`. By using the `float` data-type, instead of `int`, it makes the interface more compatible with a wider range of hardware, having different resolutions. In our implementation of the brightness-functions, the MCU's timer peripherals (`TIM`) will be configured to have a 16-bit resolution. However, the actual code interfacing with the brightness-functions doesn't need to be concerned with the available resolution; it just specifies a range of `0.00` (fully off) to `100.00` (fully on).



For most applications, `int32_t` would have been preferred over `float` since it has a consistent representation and is easier to serialize. `float` is used here to make it easier to see the differences in the interface's flexibility. Also, most people tend to think of PWM as a percentage, which maps naturally to `float`.

Deciding on queue contents

When using queues as a way of passing commands to different parts of the system, it is important to think about what the queue should actually hold, instead of what might be coming *across the wire* in a physical sense. The incoming data stream often consists of payloads with headers and footers. When storing that data in a queue, the queue will often just contain the parsed payload. Also, the data stored in the queue may be in a different format than the data stream (e.g., ASCII numbers may be converted to `int` values). Using this approach can save queue space and make the queue contents easier to process when they are received. Also, using this approach can allow more flexibility if the queue is repurposed to hold data received from other physical layers.

The LED command-executor is the task `LedCmdExecution()`, in `ledCmdExecutor.c`. When the task receives a command from the queue, it calls the LEDs' brightness-functions. So, it is convenient for the queue to hold brightness-levels in a data-format that can be passed directly to the brightness functions, i.e., `float`.

The queue that holds commands is created in `mainColorSelector.c`, in `main()`. The queue's handle is `ledCmdQueue`. There are two main considerations when deciding on what data the command-queue contains:

- `ledCmdExecutor` will be dealing with the LED brightness-functions directly, so it makes sense to store floats in the queue.
- We'd also like our LED command-executor to have different modes of operation, e.g., blink. So, we will also need a way of specifying a mode in a command. We'll only have a handful of modes, so a `uint8_t` is a good fit (an 8-bit unsigned integer).

This results in the following `struct LedCmd`, for the elements in the command-queue. (It's defined in `Inc/ledCmdExecutor.h`.) The variable `cmdNum` specifies the mode, and it will be represented by an `enum` (shown later).

```
typedef struct
{
    uint8_t cmdNum;
    float red;
    float green;
    float blue;
} LedCmd;
```

So, the LED command-executor's primary input-interface will be a queue with elements of type `LedCmd`.



Since this structure is only 13 bytes, we'll simply pass it by value. Passing by reference (a pointer to the structure) would be faster, but it also complicates the ownership of the data. These trade-offs are discussed in *Chapter 9*, *Intertask Communication*.

Now that we have the command-queue's data-model defined, we can look at the remaining components of this system.

Defining the architecture

The command-executor architecture is composed of three primary blocks. Each block executes asynchronously relative to the others, and they communicate via a queue and stream-buffer:

- **LED command-executor**: The task `LedCmdExecution` receives commands from the queue `ledCmdQueue`, and it actuates the LEDs using the LED brightness-functions. When the task is created, it is passed an argument of type `CmdExecArgs`. The task is declared in `ledCmdExecutor.c`, and it is created by `main()` in `mainColorSelector.c`.
- **Frame protocol-decoding**: The USB driver receives data sent to the MCU's USB connection, from the development computer. The received data is put in a stream-buffer. The task `frameDecoder()` receives the data in the stream-buffer. `frameDecoder()` ensures valid command framing, and puts the commands in the queue `ledCmdQueue`. The task `frameDecoder()` is created by `main()` in `mainColorSelector.c`, and the task is also declared there.

- **The USB driver**: The USB library is spread across many files. Our USB driver is primarily implemented in `vcpDriverMultiTask.c`. The USB driver's send features were presented in *Chapter 12*. Its receive features are used in this system.

Here's a visual representation of how all of these major components stack up and flow together. Major blocks are listed on the left, while representations of the data they operate on are to the right:

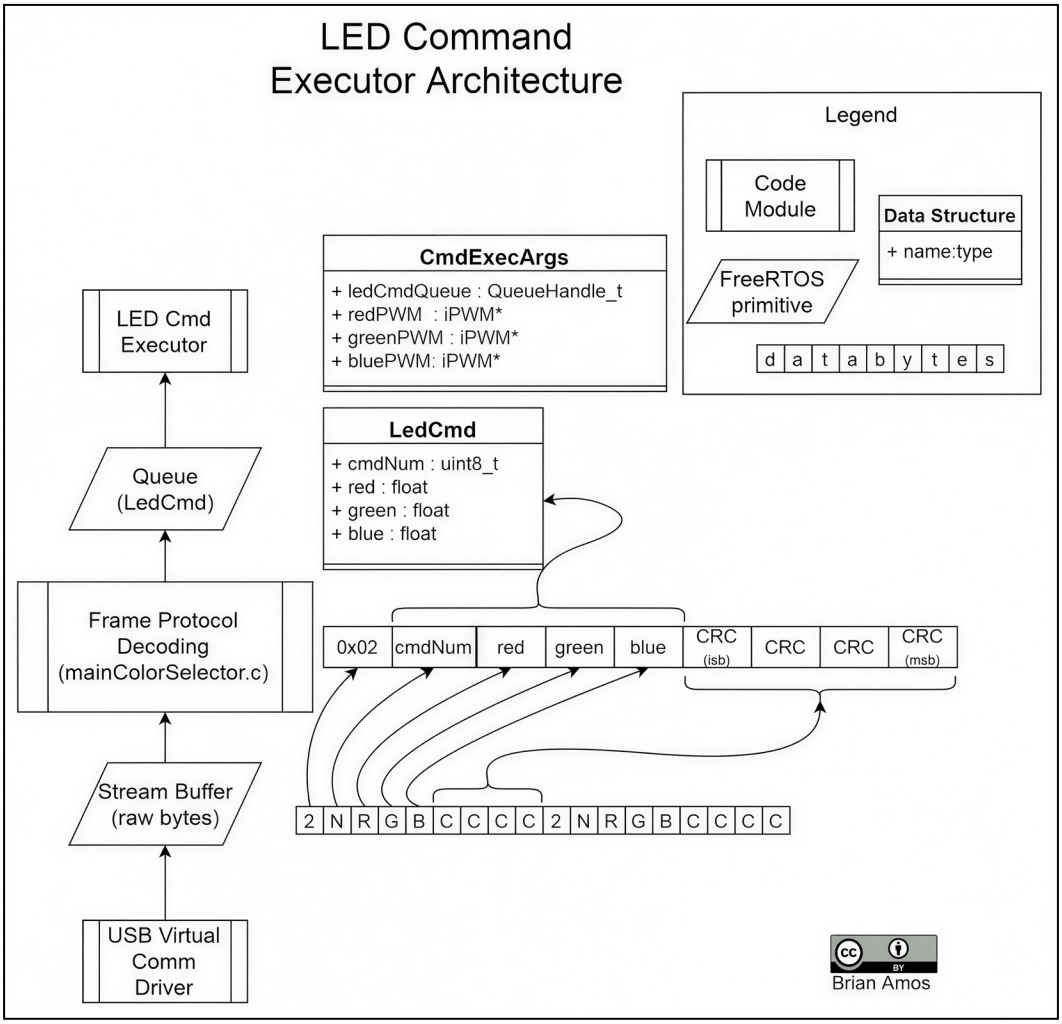


Figure 13.2 : LED command-executor architecture

Let's take a closer look at each of these components.

The command-execution task

The task `LedCmdExecution` implements a simple state machine, whose state is modified when a command is received from the queue.

The available commands are explicitly enumerated by `LED_CMD_NUM`. (It's in `Inc/ledCmdExecutor.h`.) Each command has been given a human-friendly enumeration, along with an explicit definition. The enums are explicitly defined so they can be properly enumerated on the PC side. We also need to make sure the numbers assigned are less-than-or-equal-to 255, since we'll only be allocating 1 byte to the command-number, in the frame:

```
typedef enum
{
    CMD_ALL_OFF = 0,
    CMD_ALL_ON = 1,
    CMD_SET_INTENSITY = 2,
    CMD_BLINK = 3
} LED_CMD_NUM;
```

The task `LedCmdExecution` takes an argument:

```
void LedCmdExecution(void* Args)
```

`void* Args` actually has a type of `CmdExecArgs`. (It's in `Inc/ledCmdExecutor.h`.) However, the function signature for a FreeRTOS task requires a single parameter of `void*`. The actual data type being passed into `LedCmdExecution` is a pointer to this struct:

```
typedef struct
{
    QueueHandle_t ledCmdQueue;
    iPWM * redPWM;
    iPWM * bluePWM;
    iPWM * greenPWM;
} CmdExecArgs;
```

Passing in references to the task allows multiple instances of the task to be created and run simultaneously. It also provides extremely loose coupling to the underlying `iPWM` implementations.

`LedCmdExecution` has a few local variables to track state:

```
LED_CMD_NUM currCmdNum = CMD_ALL_OFF;
bool blinkingLedsOn = false;
LedCmd nextLedCmd;
. . .
CmdExecArgs args = *(CmdExecArgs*)Args;
```

Let's take a closer look at these variables:

- `currCmdNum`: Local storage for the current command being executed.
- `blinkingLedsOn`: Local storage used by the `blink` command to track state.
- `nextLedCmd`: Storage for the next command coming from the queue.
- `args`: A local variable containing the arguments passed in through our task's parameter `void* Args`. Notice the explicit type-cast.

To ensure none of the pointers change, we're making a local copy of the parameter `Args`. To save a bit of space, this also could have been accomplished by defining the `CmdExecArgs` struct to contain `const` variables that could only be set at initialization.

In the task `LedCmdExecution`, the primary while-loop has two responsibilities. As shown in the following code, the primary while-loop: a) copies a value from `ledCmdQueue` into the `nextLedCmd`, and b) sets the appropriate local variables, and the duty-cycle of the LEDs.

```
while(1)
{
    if(xQueueReceive(args.ledCmdQueue, &nextLedCmd, 250) == pdTRUE)
    {
        switch(nextLedCmd.cmdNum)
        {
            case CMD_SET_INTENSITY:
                currCmdNum = CMD_SET_INTENSITY;
                setDutyCycles( &args, nextLedCmd.red,
                               nextLedCmd.green, nextLedCmd.blue);
                break;
            case CMD_BLINK:
                currCmdNum = CMD_BLINK;
                blinkingLedsOn = true;
                setDutyCycles(&args, nextLedCmd.red,
```

```
                                nextLedCmd.green, nextLedCmd.blue);
                break;
            . . .
            //additional cases not shown
        }
    } // END OF: if(xQueueReceive. . .
```

The second part of the main loop, seen in the following code, executes if no command has been received from `ledCmdQueue` within 250 ticks (250 ms, since our configuration uses a 1 kHz tick). This code toggles the LEDs between their last commanded duty-cycle and `OFF`:

```
    else if (currCmdNum == CMD_BLINK)
    {
        // If there is no new command and we should be blinking
        if(blinkingLedsOn)
        {
            blinkingLedsOn = false;
            setDutyCycles(&args, 0, 0, 0);
        }
        else
        {
            blinkingLedsOn = true;
            setDutyCycles(&args, nextLedCmd.red,
                          nextLedCmd.green, nextLedCmd.blue);
        }
    }
```

Finally, the `setDutyCycles` helper function uses the `iPWM` pointers to actuate the PWM duty-cycles for the LEDs. The `iPWM` pointers were verified as not being `NULL` before the task's primary while-loop, so the check doesn't need to be repeated here:

```
void setDutyCycles( const CmdExecArgs* Args, float RedDuty,
                    float GreenDuty, float BlueDuty)
{
    Args->redPWM->SetDutyCycle(RedDuty);
    Args->greenPWM->SetDutyCycle(GreenDuty);
    Args->bluePWM->SetDutyCycle(BlueDuty);
}
```

That wraps up the high-level functionality of our LED command-executor. The main purpose of creating a task like this was to illustrate a way to create an extremely loosely coupled and scalable system. While it is silly to toggle a few LEDs in this way, this design paradigm is perfectly scalable to complex systems and capable of being used on different hardware without modification.

Now that we have an idea of what the code does at a high level, let's take a look at how the `LedCmd` struct is populated.

Frame decoding

As data comes in from the USB connection, it is placed in a stream-buffer via the USB library and utility functions.

The USB's receiving stream-buffer is created by `vcpInit()`, which is part of our USB driver `vcpDriverMultiTask.c`. `vcpInit()` is called by `main()` in `mainColorSelector.c`. The stream-buffer's handle is defined as a global variable in `vcpDriverMultiTask.c`. Users of the driver can access the handle via the function `GetUsbRxStreamBuff()`, which is shown below. (It's in `vcpDriverMultiTask.c`.)

```
StreamBufferHandle_t const * GetUsbRxStreamBuff( void )
{
    return &vcom_rxStream;
}
```

This function returns a constant pointer to a stream-buffer handle of type `StreamBufferHandle_t`. This is done so the calling code can access the stream-buffer directly, but isn't able to change the pointer's value.

The protocol for the commands is a binary stream that starts with 0x02 and ends with a CRC-32 checksum, transmitted in little-endian byte-order:

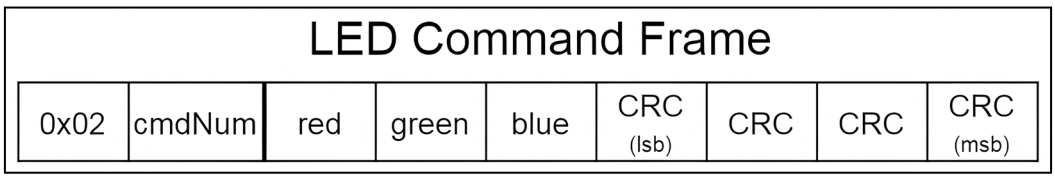


Figure 13.3: The bytes in an LED command-frame

There are many different ways of serializing data. A simple binary stream was chosen here for simplicity. A few points should be considered:

- The `0x02` header is a convenient delimiter that can be used to find the (possible) start of a frame. It is not sufficiently unique since any of the other bytes in the message can also be `0x02` (it is a binary stream, not ASCII). The CRC-32 at the end provides assurance that the frame was correctly received.
- Since the frame has exactly 1 byte per LED value, we can represent the 0–100% duty cycle with 0–255 (i.e., 255 is 100%). This way, we are guaranteed to have valid and in-range LED values, without any additional checking.
- This simple method of framing is extremely rigid and provides no flexibility whatsoever. The moment we need to send something else over the wire, we're back to square one. A more flexible (and complex) serialization method is required if flexibility is desired.

The `frameDecoder()` task is defined in `mainColorSelector.c`, and shown below. It loops to get command-frames from the USB's receiving stream-buffer. A command-frame is decoded and put in a `LedCmd` struct. The `LedCmd` struct is put in the command-queue.

```
void frameDecoder( void* NotUsed)
{
    LedCmd incomingCmd;
#define FRAME_LEN 9
    uint8_t frame[FRAME_LEN];
    while(1)
    {
        memset(frame, 0, FRAME_LEN);
        while(frame[0] != 0x02)
        {
            xStreamBufferReceive( *GetUsbRxStreamBuff(), frame, 1,
                                  portMAX_DELAY);
        }
        xStreamBufferReceive( *GetUsbRxStreamBuff(),
                              &frame[1],
                              FRAME_LEN-1,
                              portMAX_DELAY);
        if(CheckCRC(frame, FRAME_LEN))
        {
            incomingCmd.cmdNum = frame[1];
```

```
            incomingCmd.red = frame[2]/255.0 * 100;
            incomingCmd.green = frame[3]/255.0 * 100;
            incomingCmd.blue = frame[4]/255.0 * 100;
            xQueueSend(ledCmdQueue, &incomingCmd, 100);
        }
    }
}
```

Let's break it down line by line:

- Two local variables are created, `incomingCmd` and `frame`. `incomingCmd` is a `struct` that is used to store the fully parsed command. `frame` is a buffer of bytes that is used to store exactly one frame's worth of data, while the present function parses and verifies it:

  ```
  LedCmd incomingCmd;
  #define FRAME_LEN 9
  uint8_t frame[FRAME_LEN];
  ```

- At the beginning of the loop, the contents of `frame` are cleared. Only clearing the first byte is strictly necessary, so we can accurately detect `0x02`, and since the frame is binary and has a well-defined length. (Only variable-length strings *need* to be null-terminated.) However, it is very convenient to see `0`'s for unpopulated bytes, if you happen to be watching the variable during debugging:

  ```
  memset(frame, 0, FRAME_LEN);
  ```

- A single byte is copied from the `stream-buffer` into the frame until `0x02` is detected. This should indicate the start of a frame (unless we were unlucky enough to start acquiring data in the middle of a frame that contains a `0x02` in the payload or CRC):

  ```
  while(frame[0] != 0x02)
  {
      xStreamBufferReceive( *GetUsbRxStreamBuff(), frame, 1,
                            portMAX_DELAY);
  }
  ```

- The remaining bytes of the frame are copied from the `stream-buffer`. They are placed in the correct index of the `frame` array:

  ```
  xStreamBufferReceive( *GetUsbRxStreamBuff(), &frame[1],
                        FRAME_LEN-1, portMAX_DELAY);
  ```

- The entire frame's CRC is evaluated. If the CRC is invalid, this data is discarded, and we'll begin looking for the start of another frame. (`CheckCRC()` is implemented as a stub, and always returns `True`.)

```
if(CheckCRC(frame, FRAME_LEN))
```

- If the frame was intact, the `struct incomingCmd` is filled out with the values in the frame:

```
incomingCmd.cmdNum = frame[1];
incomingCmd.red = frame[2]/255.0 * 100;
incomingCmd.green = frame[3]/255.0 * 100;
incomingCmd.blue = frame[4]/255.0 * 100;
```

- The populated command is sent to the queue, which is being watched by the task `LedCmdExecutor()`. Up to `100` ticks may elapse, waiting for space in the queue to become available before the command is discarded:

```
xQueueSend(ledCmdQueue, &incomingCmd, 100);
```



It is important to note that none of the framing protocol is being placed in `LedCmd`, which is sent to the queue – only the payload is. This allows more flexibility in how data is acquired before being queued, as we will see in the upcoming section *Reusing a queue definition for a new target*.

Choosing the number of slots available in the queue can have important effects on the response of the application to incoming commands. The more slots that are available, the higher the likelihood that a command will incur a significant delay before being executed. For a system that requires more determinism for when a command will be executed, it is good to limit the queue length to a single slot. If more determinism is needed for whether a command will be executed, a protocol-level acknowledgment can be implemented.

Now that we've seen how the frame is decoded, the only remaining piece of the puzzle is how data is placed into the USB's receiving stream-buffer.

The USB VCP driver

The USB's receiving stream-buffer is populated by `CDC_Receive_FS()` in `Drivers/HandsOnRTOS/usbd_cdc_if.c`. The function is shown below. The call to `xStreamBufferSendFromISR()` was added for the example-program. The function looks similar to the code from *Chapter 12*, where the transmit side of the driver was developed (`CDC_Transmit_FS()`). How the USB receive-process works is not described, as it's beyond the scope of the book.

```
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len)
{
    /* USER CODE BEGIN 6 */
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
    xStreamBufferSendFromISR( *GetUsbRxStreamBuff(),
                              Buf, *Len, &xHigherPriorityTaskWoken);

    USBD_CDC_ReceivePacket(&hUsbDeviceFS);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    return (USBD_OK);
    /* USER CODE END 6 */
}
```

Using a stream-buffer instead of a queue allows larger blocks of memory to be copied from the USB stack's internal buffers while providing a queue-like interface that has flexibility in the number of bytes copied out of it. This flexibility is one of the reasons coding the protocol layer was so straightforward.



Remember that since a stream-buffer is being used, only one task can be a designated reader. Otherwise, access to the stream-buffer must be synchronized (that is, by a mutex).

That wraps up all of the MCU-side code in the example-program. Since this example relies on a binary protocol over USB, let's have a look at how to send commands from the development computer to the MCU.

Using the code

One of the goals for choosing this example was to have an approachable, real-world use-case that was relevant. Most of the time, the use-cases for the embedded systems we develop do not include a person typing away on a terminal emulator. To that end, a very simple GUI was created using Python to make it simple to send commands to the dev-board. The script is `PythonColorSelectorUI/colorSelector.py`. To use it, Python must be installed (at least v3.8). The required Python packages are listed in `requirements.txt`.

A Windows binary is also included (`PythonColorSelectorUI/dist/colorSelector.exe`). The `*.exe` doesn't require Python to be installed.

To use the GUI:

1. Using Ozone, download the example-program to the dev-board and start it.
2. Start the GUI, and select the USB connection's **Virtual COM Port (VCP)**. (Selecting a VCP is described in *Chapter 12*.)

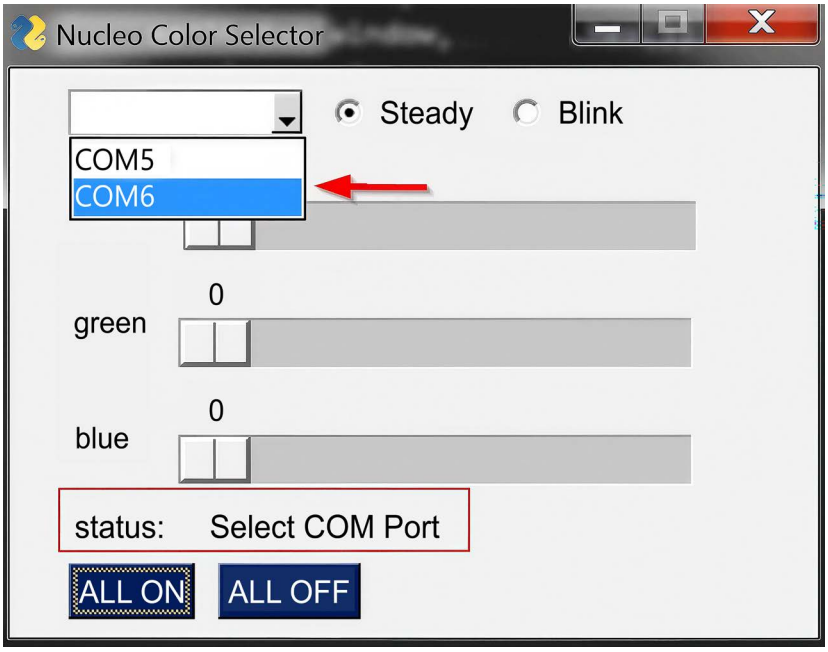


Figure 13.4: The app which sends commands

3. After the GUI has been successfully opened, use the GUI's sliders and buttons to actuate the LEDs on the dev-board. A command frame is constructed on each GUI update event, and it is immediately sent over USB to the dev-board. The last frame sent is displayed on the GUI, in ASCII hex-dump format.

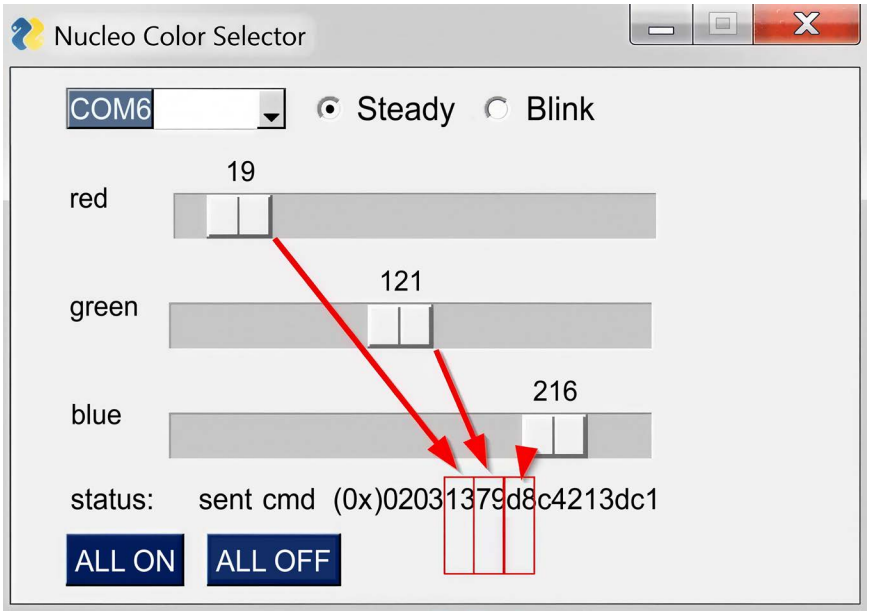


Figure 13.5 : The command-frame is shown on the GUI

As an alternative to the GUI, a terminal application capable of sending binary-data could also be used. Tera Term is an example for Windows, and a macro script can be used to send binary-data. For example, this macro instruction turns on all LEDs:

```
send$02$01$00$00$00$00$00$00
```

So, we have some blinky lights and a (not-so) flashy UI. Let's address the real takeaway of this exercise – the flexibility we've built into our application by using a queue.

Reusing a queue definition for a new target

On the surface, it might be hard to appreciate just how flexible a setup such as this one is. On the command-entry side, we have the ability to acquire commands from anything, not just the binary framed protocol over USB. Since the data being placed into the queue was abstracted to not include any protocol-specific information, the underlying protocol could change without requiring any changes downstream.

Let's take a look at a few examples:

- We could write a different routine for parsing incoming data that uses a comma-separated ASCII string, where duty cycles are represented by percentages between 0 and 100, and the string is terminated by a new line: `BLINK,20,30,100\n`. This would result in the following value being placed in `ledCmdQueue`:

```
LedCmd cmd = {.cmdNum=3, .red=20, .blue=30, .green=100};
xQueueSend(ledCmdQueue, &cmd, 100);
```

- The underlying interface could change completely, e.g., from USB to UART, SPI, I2C, Ethernet, etc.).
- The commands could come in the form of a non-serialized data source (discrete duty-cycles or physical pins of the MCU, for example).

There is also no reason for a queue to be limited to being populated by a single task. If it makes sense from a system-design perspective, a command executor's queue could be populated by any number of simultaneous sources (shown in *Figure 13.6*). The possibilities of how to get data into the system are truly endless, which is great news – especially if you're developing something considerably more complex, that will be used in multiple systems. You're free to invest time in writing quality-code once, knowing that the code will be used in multiple applications because it is flexible enough to adapt, while still maintaining a consistent interface.

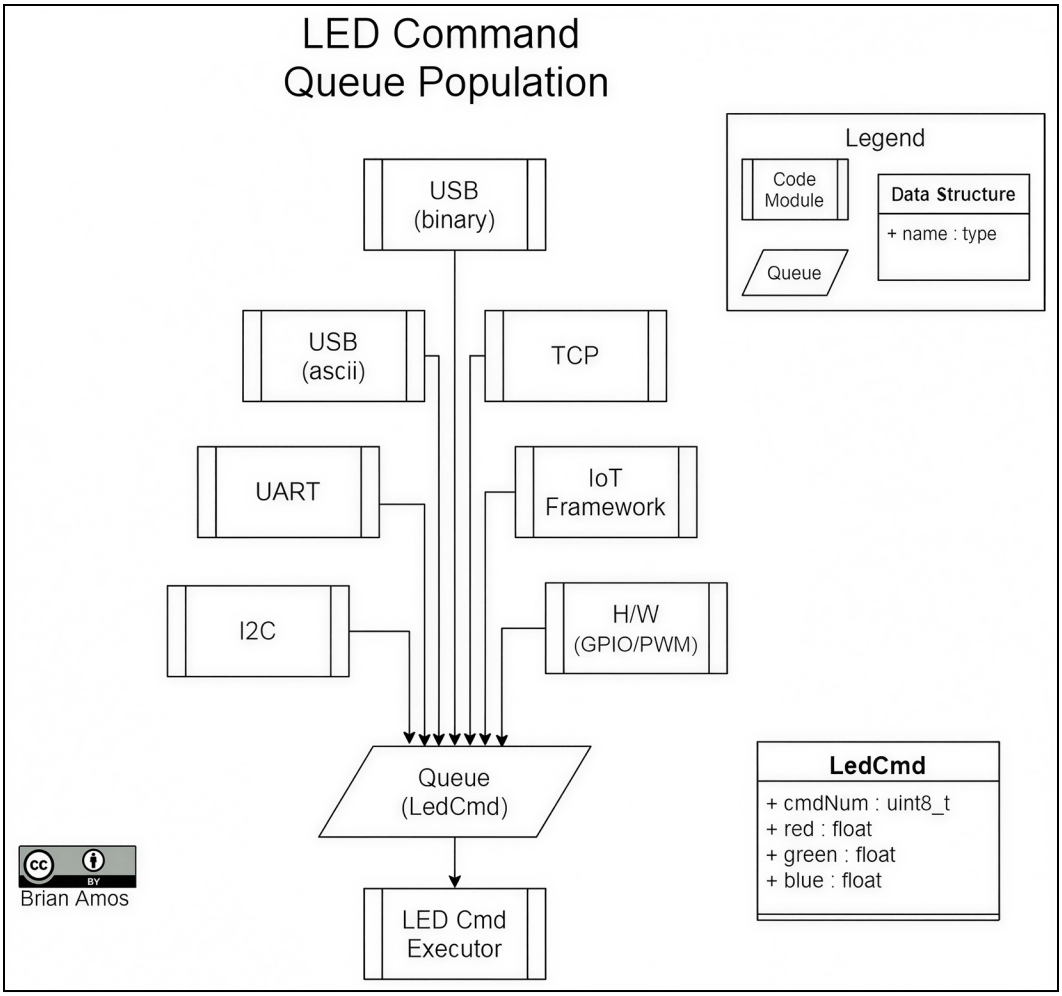


Figure 13.6: Populating the LED command-queue

For the LED command-executor, there are two components that provide flexibility – the queue interface and the `iPWM` interface. We'll consider them next.

The queue interface

After the task `LedCmdExecution()` is started, it only interacts with the upstream code through the command queue. Because of this, we are free to change the task's underlying implementation without directly affecting the upstream code (as long as the data passed through the queue has the same meaning).

For example, blinking could be implemented differently and none of the code feeding the queue would need to change. The only requirements for passing a command through the queue are using a `uint8_t` command-number and three floating-point numbers. So, we're free to completely rewrite the implementation of `LedCmdExecution` (without the `iPWM` interface, for example). This change will only affect a single file that starts the task – `mainColorSelector.c`, in this example. Any other files that deal with the queue directly will be unaffected. If there were multiple sources feeding the queue (such as a USB, I2C, IoT framework, and so on), none of them would need to be modified, even after completely changing the underlying `LedCmdExecution()` implementation.

The iPWM interface

The three functions used to set the LEDs' intensity are `SetBlueDuty()`, `SetRedDuty()`, and `SetGreenDuty()`. They are declared in `Src/pwmImplementation.c`. The functions set the intensity by using the MCU's timer and PWM features. However, to set the LED's intensity, we're free to substitute any other implementation of these functions.

For example, the LEDs could be driven by a serial interface, which requires a serial data stream rather than PWM. The LEDs may even be remotely located and require another protocol to actuate them. As long as the LED intensity can be represented by a percentage between 0 and 100, it can be controlled by these functions.

Now, realistically, in the real world, you're not likely to go to all of this trouble just to blink a few LEDs! Keep in mind that this is an example with fairly trivial functionality, so we're able to keep our focus on the actual architectural elements. In practice, flexible architectures provide a foundation for building long-living and adaptable codebases.

All of this flexibility does come with a warning – always be careful of the systems you are designing and ensure they meet their primary requirements. There are always tradeoffs to be made, whether they are between performance, initial development time, BOM cost, code elegance, flash space, or maintainability. After all, a beautifully extensible design doesn't do anyone any good if it doesn't fit in the available ROM and RAM!

Understanding queues as interfaces

In our two recent example-programs, we've seen how tasks, queues, and stream-buffers can be used to create well-structured designs for processing **serial data streams**. These designs, and their code, can be reused, as they are applicable to other serial data streams, with similar data structures and processing requirements.

Let's take a look at why using queues is a good idea. The concepts are applicable to similar data structures, such as stream-buffers:

- **Queues make excellent interface definitions:** Queues provide a very clear line of abstraction. To pass data through a queue and get a desired behavior on the other side, all of the needed data must be present. And, the sender and receiver must both have a common understanding of what the data format is. This clean line forces a level of conscientious thought as to what exactly needs to be communicated. Sometimes, this level of active thought isn't present when implementing individual functions. The delineation provided by the queue forces additional thought about what information is required, as well as its format. Responsible developers will be more likely to ensure these types of definitive interfaces are thoroughly documented.

  Also, when a queue is viewed as an interface to a subsystem, it pays dividends to document the functionality that will be provided, as well as the exact formatting required to use the subsystem. Often, the fact that an interface is well defined will increase the likelihood of reuse since it will be easily understood.

- **Queues increase flexibility:** Queues are beautiful in their simplicity – a sender places something in the queue and whatever task is monitoring the queue will receive the data and act on it. The only things the sending and receiving tasks need to share are the specifications for the queue's size and data-format. Since the list of shared resources is so short, there is a natural decoupling effect when queues are used.

  Because of the clean break provided by the queue, the exact implementation of the sending and receiving tasks could change over time. The same functionality can be implemented in many different ways in one task, which won't affect the other task, as long as the queue-interface doesn't change.

It also means that data can be sent to the queue from anywhere; there are no explicit requirements for the physical interface – only the data format. This means that a queue can be designed to receive the same data stream from many different interfaces, which can provide system-level flexibility. The functionality doesn't need to be tied to a specific physical interface, such as Ethernet, USB, UART, SPI, CAN, etc.

- **Queues make testing easier:** Queues provide excellent places to enter test data. This provides a very convenient entry-point for inserting test data for code that is under development. For example, the task that sends data to the queue can be simulated by generating test data and sending it to the queue. The task that receives data from the queue can be tested with this test data.

Summary

In this chapter, you have gained first-hand experience in creating a simple end-to-end command-executor architecture. At this point, you should be quite comfortable with creating queues, and using tasks to send to queues and receive from queues. And, you will have started to gain a deeper understanding of how tasks and queues can be used to process serial data streams. Also, we've seen the advantages queues provide as clean interfaces between tasks. You can apply variations of these techniques to many real-world projects. If you're feeling particularly adventurous, feel free to implement one of the suggested protocols or add an entry point for another interface (such as a UART).

In the next chapter, we'll change gears and discuss some of the low-level details of memory allocation.

Further reading

For information on PWM, here are some useful tutorials and manuals, from ST:

"STM32L4 - Timers : Advanced-control, general-purpose, and basic timers," Revision 2.0, undated

- `https://www.st.com/resource/en/product_training/STM32L4_WDG_TIMERS_GPTIM.pdf`
- `https://community.st.com/s/question/0D53W00001Gia5QSAR/an-stm-tutorial-on-timers-has-errors-regarding-pwm-mode-for-stm32-mcus`
- "Counting period management," page 6
- PWM calculations, pages 31-33
- Dimmable LEDs, pages 31-33

"STM32 cross-series timer overview," Application note AN4013, June 2021

- Section 2.5, "Timer in PWM mode," page 15f
- On page 16, this phrase has an ambiguity, "To configure the timer in this mode:". Apparently, "This mode" refers to PWM-mode as a whole, not to just PWM mode 2.

"Reference manual : STM32F76xxx and STM32F77xxx advanced Arm-based 32-bit MCUs," RM0410, March 2018

- Section 26.3.9, "PWM mode," pages 985f

Part 4

System Design and Troubleshooting

In Part 4 of the book, two system design topics are presented for FreeRTOS-based systems: FreeRTOS memory management and FreeRTOS use with multi-processor and multi-core systems. In addition, FreeRTOS troubleshooting is presented, including debugging tools and debugging techniques.

Part 4 includes the following chapters:

- *Chapter 14*, *FreeRTOS Memory Management*
- *Chapter 15*, *Multi-Processor and Multi-Core Systems*
- *Chapter 16*, *Troubleshooting Tips and Next Steps*

14

FreeRTOS Memory Management

So far, we've worked through many examples of creating FreeRTOS primitives, such as tasks, semaphores, queues, etc. However, when these primitives were initially created, there wasn't much of an explanation as to where the memory was coming from. In this chapter, we'll learn exactly where the memory comes from, along with when and how it is allocated. Choosing when and how memory is allocated allows us to make trade-offs between coding convenience, timing determinism, potential regulatory requirements, and code standards. We'll conclude by looking at different measures that can be taken to ensure application robustness.

The chapter has three example-programs, and they are in the IDE-project `Chapter_14`.

In a nutshell, this chapter covers the following:

- Understanding memory allocation
- Static and dynamic allocation of FreeRTOS primitives
- FreeRTOS heap implementations
- Replacing `malloc` and `free`
- Monitoring for stack and heap problems
- Using a **memory protection unit (MPU)**

Understanding memory allocation

Memory allocation isn't necessarily at the top of a developer's list of favorite topics to consider when developing an application – it just isn't all that glamorous. **Dynamic memory-allocation** has to do with allocating memory as it is needed during runtime, rather than during compile time. With desktop applications, memory is generally available whenever it is needed, so it isn't given a second thought; memory is simply a `malloc` call away. And, when the memory is no longer needed, it will be unallocated with `free`.

Unlike the carefree dynamic memory-allocation schemes in a desktop environment, embedded-systems programmers typically must be more careful about how memory is dynamically allocated. In an embedded system, there's typically less memory available, and there are additional memory-related requirements, such as regulatory requirements and timing constraints.

Many high-reliability and safety-critical coding standards do not allow the use of dynamic allocation, such as MISRA-C. In these cases, it is still perfectly acceptable to use static-allocation. Some coding standards disallow dynamic allocation after the tasks are created (for example, *Ten Rules for Safety Critical Coding* by Gerard Holzmann). In this case, static allocation would be reasonable, or FreeRTOS's `heap_1.c` implementation.`

RAM may be severely limited on a given platform. On the surface, this seems like the perfect use-case for dynamic memory-allocation; after all, if there is limited memory, it can be given back when it's not in use! In practice, however, things don't always go this smoothly when there is limited heap-space available. When small heaps are used to allocate space for arbitrarily sized objects with different lifetimes, then memory fragmentation often occurs, eventually.

Finally, a need for highly deterministic timing can also limit the options for dynamic allocation. If a portion of code has tight timing constraints, it is sometimes easier to avoid using dynamic allocation, rather than contriving tests that attempt to mimic worst-case timing for a call to `malloc`. It is also worth noting, again, that `malloc` isn't guaranteed to succeed, *especially* on an embedded system with limited memory. In a memory-constrained and multi-threaded system, having a large amount of dynamic allocation can create complex use-cases that potentially fail at runtime. And, thoroughly testing such a system is a serious challenge.

With this information on why memory allocation is so important in embedded systems, let's take a closer look at *where* memory comes from: static memory, stack memory, and heap memory.

Static memory

Static memory includes global variables, and variables declared inside of functions using the `static` specifier. Static-memory's lifespan is the entire duration of a program.

For example, in the following code-snippet, both `globalVar` and `staticVar` are located in static memory, and they will persist for the entire lifetime of the program. Their initialization occurs only once, during the initial program load.

```
uint8_t globalVar = 12;

void myFunc( void )
{
    static uint8_t staticVar = 0;
    ...
}
```

When variables are declared as static, memory is guaranteed to be available. All of the global and static variables defined by the program are placed into their locations during the linking phase. If all of the available memory has been properly configured (e.g., memory on the MCU, and external to the MCU), the linker guarantees that space is available for these variables.

The downside to static variables is that they will *always* be consuming space, even when they are not being used.

Stack memory

Each time a function is called, data for that function is placed onto a **stack**, including the parameters and local variables. When the function exits, all of the information that was placed onto the stack is removed. (This is why passing pointers to local variables can be a problem.) In FreeRTOS, each task has its own private stack whose size is determined when the task is created.

Since stacks have such an orderly way of being accessed, it isn't possible for them to become fragmented, like a heap. It is possible, however, to overflow a stack by placing more information onto it than its size allows.



On the Cortex-M, there is also one additional stack – the **main stack**. The main stack is used by ISRs and the FreeRTOS kernel. They execute in a privileged mode that modifies the **main-stack pointer (MSP)**. On the other hand, tasks execute on the **process-stack** and use the **process-stack pointer (PSP)**. All of the stack-pointer operations are taken care of by the hardware and the kernel, depending on what is currently being executed: the kernel, ISR, or task (process). The stack-pointer operations are not something that FreeRTOS users will normally need to worry about.

Initialization of the stack and heap takes place in `Chapter_*\startup\startup_stm32f767xx.s`. The exact size of the main stack is defined in the linker script `Chapter_*\STM32F767ZI_FLASH.ld`. If necessary, the size of the stack and heap available to the system before the FreeRTOS scheduler is started can be adjusted by modifying `_Min_Heap_Size` or `_Min_Stack_Size`:

```
_Min_Heap_Size = 0x200; /* required amount of heap */`
_Min_Stack_Size = 0x400; /* required amount of stack */
```

It is best to keep both of these to minimal sizes since any RAM used here will be unavailable to the tasks. These stacks and heaps are for code that is run before the scheduler is started, and for the ISRs. This is *not* the same stack that is used by the tasks.



Occasionally, you may run into a problem where you'll need to run some memory-intensive system-initialization code (the USB stack is a good example of this). If the system-initialization functions are called outside of a task (before the scheduler starts), then these functions will use the main stack. In order to keep the main stack as small as possible and to allow more memory for tasks, move memory-intensive system-initialization into a task. This can allow the RTOS heap to have additional RAM that would have gone unused after system-initialization.

The FreeRTOS kernel manipulates the **process-stack pointer (PSP)** to point to the stack of the task that has context (i.e., in the running state).



For the most part, you won't need to be immediately concerned with the various stack pointers – they are taken care of by the kernel and C runtime. If you happen to be developing code that will transition between an RTOS and *bare metal* (e.g., a bootloader), then you will need to understand how/when to properly switch the current stack-pointer.

The most important thing to keep in mind with stacks is that they must be adequately sized to hold all of the local variables that a task will execute for the deepest call level. We'll discuss ways of getting a handle on this later, in the section *Keeping an eye on stack space*.

Heap memory

The **heap** is the portion of memory that is used when a dynamic-allocation function such as `malloc` is called. It is also where a task's stack and **task control block (TCB)** are stored when the task is created via `xTaskCreate()`.

In an MCU FreeRTOS system, there will typically be two heaps created:

- **System heap**: Defined in the startup and linker scripts described previously. This heap will *not* be available for use by the final application code when allocating space for FreeRTOS primitives.
- **FreeRTOS heap**: Used when creating tasks and other primitives. It is defined in `Inc\FreeRTOSConfig.h`. It can be resized by adjusting the following line:

```
#define configTOTAL_HEAP_SIZE ((size_t)15360)
```

Currently, that line is defining a 15 KB heap. This heap must be adequately sized to accommodate the following:

- Stacks and **TCBs** for all tasks that are created using `xTaskCreate()`
- Queues, semaphores, mutexes, event groups, and software timers created using FreeRTOS `x*Create` functions

(FreeRTOS supports 5 different types of heaps, which will be explained later. `configTOTAL_HEAP_SIZE` is used by all of them, except `heap_3`.)

Here's a visual representation of where all of the different variables will come from:

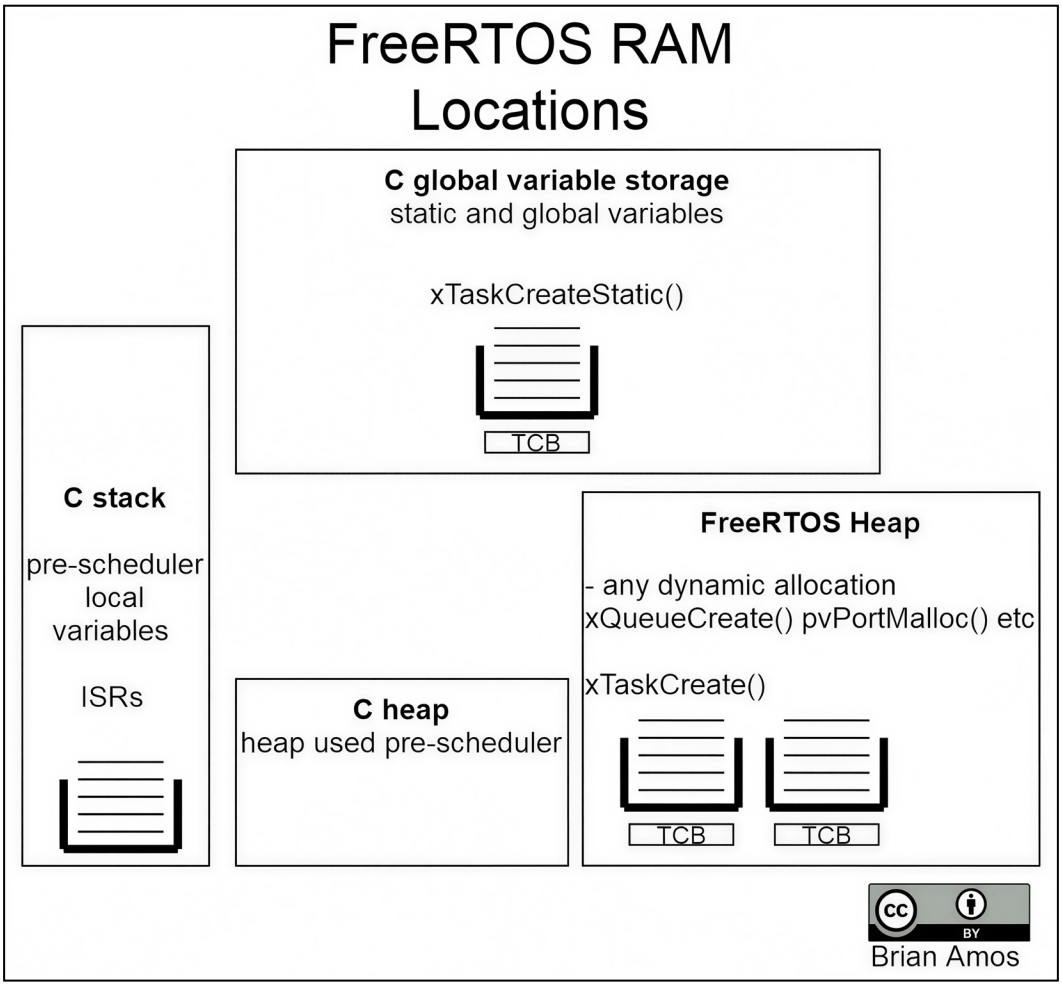


Figure 14.1: FreeRTOS memory-allocation locations

There are two possible locations for tasks' stacks and TCBs:

- Statically allocated space for a stack and a `TCB`, passed to a task when calling **xTaskCreateStatic().**
- Dynamically allocated space for a stack and a `TCB`, obtained from the heap when calling **xTaskCreate()**.

The C-heap is only used for items that are created without the use of the FreeRTOS heap implementation. And, the C-stack is only used before the scheduler is started, as well as by ISRs. When using an RTOS, it is best to minimize the size of the C-heap as much as possible, or entirely. This will leave more available RAM to allocate to the RTOS heap and to static variables.

Heap fragmentation

In an embedded system with limited RAM, **heap fragmentation** can be a very serious issue. A heap becomes fragmented when items are loaded into the heap and then removed at different points in time. The problem is that if many of the items being removed are not adjacent to one another, a larger contiguous region might not be available:

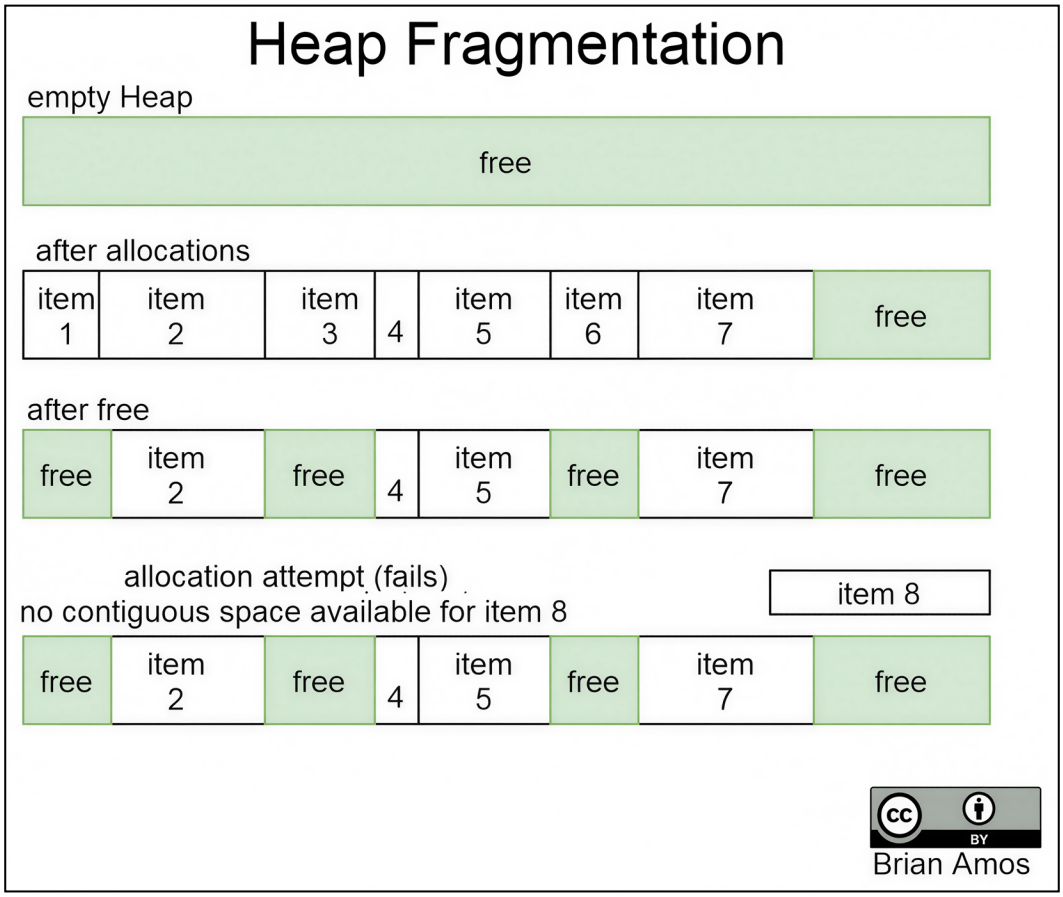


Figure 14.2: Heap fragmentation

In the preceding example, space won't be successfully allocated for item 8. Even though there is sufficient free space, there isn't enough *contiguous* free space to accommodate the size of item 8. This is especially problematic because it will only occur at runtime, and under certain circumstances that are dependent on the size of items and the timing of when items are allocated and freed.

Now that we've covered the basics of memory allocation, let's look at some different ways that FreeRTOS primitives can be created, and use static or heap memory.

Static and dynamic allocation of FreeRTOS primitives

Details on creating tasks were covered in *Chapter 7, Running the FreeRTOS Scheduler*. For the API `xTaskCreate()`, storage for the task's stack and `TCB` are taken from the FreeRTOS heap. For `xTaskCreateStatic()`, the caller provides pointers to the storage for the stack and TCB, respectively, and the stack's length is provided.

Many FreeRTOS APIs have versions for both static and dynamic allocation of stack space, such as `xTaskCreateStatic()` and `xQueueCreate()`. All FreeRTOS API functions with `*CreateStatic` in their names take additional arguments for specifying the pre-allocated memory. The memory passed to `*CreateStatic` variants will typically be statically allocated buffers, which are present for the entire program's lifetime.

In this section, we'll look at static and dynamic allocation of the stack used by FreeRTOS primitives. We'll focus on the differences in *where* the memory is coming from and what the memory's *lifetime* is. To avoid memory corruption, you'll need to have detailed knowledge of the lifetime of the FreeRTOS primitive being created and the memory allocated for it!



While the naming of the `*CreateStatic` API variants suggests that the memory is static, this isn't actually a requirement. For example, a function could allocate a buffer on its stack and pass the buffer's pointer to a `*CreateStatic` API call. However, you'll need to be sure that the lifetime of the primitive created is limited to the lifetime of that function.

Dynamic allocation examples

In nearly all of the book's FreeRTOS example-programs, the code presented has used dynamic allocation to create FreeRTOS primitives (tasks, queues, mutexes, and so on). Here are two examples to serve as a quick refresher before we look at the differences with creating primitives using static allocation.

Creating a task

When a task is created using dynamically allocated memory, the call will look something like this:

```
BaseType_t retVal = xTaskCreate( Task1, "task1", StackSizeWords, NULL,
                                 tskIDLE_PRIORITY + 2, tskHandlePtr);
assert_param(retVal != pdPASS);
```

There are a few pieces of information relevant to memory allocation, to note about this call:

- The call to `xTaskCreate` may fail. This is because there may not be enough space available for storing the task's stack and `TCB` on the FreeRTOS heap. The only way to ensure that it was created successfully is to check the return value, `retVal`.
- The only parameter having to do with a stack is the requested size of the stack, `StackSizeWords`.

When created in this manner, if it is appropriate for the task to be terminated, `xTaskDelete()` can be called. This will free the memory associated with the task's stack and TCB, so that it may be reused.

The following are a few points to note regarding dynamic allocation:

- Primitive creation may fail at runtime if no heap-space is available.
- All memory that FreeRTOS allocates for the primitive will be automatically freed when the task is deleted (as long as `Heap_1` is not used, and `INCLUDE_vTaskDelete` is set to `1` in `FreeRTOSConfig.h`). This doesn't include memory that was dynamically allocated by *user-supplied* code in the actual task; FreeRTOS is unaware of any dynamic-allocation initiated by user-supplied code. It is up to you to free this code when appropriate.
- `configSUPPORT_DYNAMIC_ALLOCATION` must be set to `1` in `FreeRTOSConfig.h`, for dynamic allocation to be available.

When creating a task using dynamic allocation, all of the memory used for the task (the task's stack and `TCB`) is allocated from the FreeRTOS heap. This was shown earlier in *Figure 14.1*.

Next, let's take a look at creating a queue.

Creating a queue

Creating queues using dynamically allocated memory was shown earlier in *Chapter 9,Intertask Communication*. As a quick review, to create a queue of length `LED_CMD_QUEUE_LEN`, with elements of type `uint8_t`, we'd go through the following steps:

1. Create the queue:

```
ledCmdQueue = xQueueCreate(LED_CMD_QUEUE_LEN, sizeof(uint8_t));
```

2. Verify that the queue was created successfully by checking that the handle `ledCmdQueue` is not `NULL`:

```
assert_param(ledCmdQueue != NULL);
```

We've reviewed examples of dynamic allocation, which allocates memory during runtime from the FreeRTOS heap. Next, let's move on to static allocation, which reserves memory during compilation and linking, before the application is ever run.

Static allocation examples

FreeRTOS also has a method for creating primitives that don't require us to dynamically allocate memory. This is an example of creating primitives with statically allocated memory.

Creating a task

To create a task using a pre-allocated stack and TCB (requiring no dynamic-allocation), use a call similar to the following:

```
StackType_t GreenTaskStack[STACK_SIZE];
StaticTask_t GreenTaskTCB;
TaskHandle_t greenHandle = NULL;
greenHandle = xTaskCreateStatic( GreenTask, "GreenTask", STACK_SIZE,
                                 NULL, tskIDLE_PRIORITY + 2,
                                 GreenTaskStack, &GreenTaskTCB);
assert_param( greenHandle != NULL );
```

There are several notable differences between this static-allocation and the previous method of dynamic allocation:

- When `xTaskCreateStatic()` runs successfully, it returns a task handle (`greenHandle`), instead of a return-value of `pdPASS`.

- Task-creation using `xTaskCreateStatic()` will always succeed, provided that the stack pointer and `TCB` pointer are non-null.
- As an alternative to checking `greenHandle`, we could check `GreenTaskStack` and `&GreenTaskTCB` instead. As long as they are not `NULL`, the task will be successfully created.
- Tasks created with `xTaskCreateStatic()` can be *deleted*. FreeRTOS will only take the steps necessary to remove the task from the scheduler. Freeing the associated memory is the responsibility of the caller.

When creating a task using static allocation, all of the memory used for the task is provided by the caller (the task's stack and `TCB`). The memory is typically allocated in the C global-variable storage-area. This was shown earlier in *Figure 14.1*.

Static creation allows more flexibility in memory-allocation than the name implies. After calling `vTaskDelete()` for a statically-created task, FreeRTOS will no longer access the memory from that task's stack or `TCB`. It is then safe to repurpose this memory for other purposes. This is illustrated in the example-program `main_staticTask_Delete.c`. There, a task is created by using `xTaskCreateStatic()`, then the task is deleted by using `vTaskDelete()`.

Creating a queue

Now let's take a look at the steps for creating a queue using static memory for both the queue contents and the queue data-structure. The following code is from the example-program `mainStaticQueueCreation.c`, and it is explained below.

```
static QueueHandle_t ledCmdQueue = NULL;
static StaticQueue_t queueDataStructure;
#define LED_CMD_QUEUE_LEN 2
static uint8_t queueStorage[LED_CMD_QUEUE_LEN];
ledCmdQueue = xQueueCreateStatic(LED_CMD_QUEUE_LEN, sizeof(uint8_t),
                                 queueStorage, &queueDataStructure );
assert_param(ledCmdQueue != NULL);
```

1. A variable is defined for holding the queue data structure used by FreeRTOS:

   ```
   static StaticQueue_t queueDataStructure;
   ```

2. A raw array is created, appropriately sized to hold the queue contents:

   ```
   static uint8_t queueStorage[LED_CMD_QUEUE_LEN];
   ```

- A simple C array of the target data type can be used; in this case, our queue will hold a data-type of `uint8_t`.
- Use `#define` to define the array length:

```
#define LED_CMD_QUEUE_LEN 2
```

3. Create the queue with the same length as the array that was previously defined:

```
ledCmdQueue = xQueueCreateStatic( LED_CMD_QUEUE_LEN, sizeof(uint8_t),
                                  queueStorage, &queueDataStructure );
```

- Here's a breakdown of the parameters:
  - `LED_CMD_QUEUE_LEN`: Number of elements in the queue
  - `sizeof(uint8_t)`: Size of each element (in bytes)
  - `queueStorage`: Raw array used for storing elements in the queue (used only by FreeRTOS)
  - `&queueDataStructure`: Pointer to a `StaticQueue_t`, the queue data-structure used internally by FreeRTOS

4. `xQueueCreateStatic()` returns a queue handle. Ensure the queue was created by verifying the returned queue-handle is not `NULL`. Though, unlike dynamically-allocated queues, it is unlikely that a call to `xQueueCreateStatic()` will fail.

```
assert_param(ledCmdQueue != NULL);
```

The only difference between creating queues with static allocation and dynamic allocation is how the memory is supplied – both calls return queue handles. Now that we've seen examples of creating queues and tasks without using dynamically allocated memory, what happens if we have a requirement for the system to have *no* dynamically allocated memory, at all?

Eliminating all dynamic allocation

In the book's example-programs, when creating FreeRTOS primitives, we've focused on using dynamic allocation APIs. This has been primarily for ease of use and brevity. It enabled us to focus on the core FreeRTOS concepts rather than worrying about exactly where memory was coming from and how we were accessing it.

All FreeRTOS primitives can be created with either dynamically allocated memory or pre-allocated memory. To avoid dynamic allocation, simply use the `*CreateStatic` version of an API's `create-` functions. `*CreateStatic` versions exist for tasks, queues, mutexes, semaphores, stream buffers, message buffers, event groups, and timers. These APIs share the same arguments as their dynamic counterparts, but also require a pointer to pre-allocated memory to be passed to them. The `*CreateStatic` equivalents don't require any memory allocation to take place during runtime.

You would consider using the static equivalents for the following reasons:

- They are guaranteed to never fail due to being unable to allocate memory.
- All of the checks that are needed to ensure memory is available happen during linking (before the application's binary is created). If memory is not available, it will fail at link time, rather than runtime.
- Many standards that target safety-critical applications prohibit the use of dynamically allocated memory.
- Internal C coding-standards will occasionally prohibit the use of dynamic allocation, for embedded systems.



Memory fragmentation could be added to this list as well, but this isn't an issue unless memory is freed. For example, `heap_1` (described later) could be used to eliminate heap-fragmentation concerns.

Now that we have an understanding of the differences between dynamic and static allocation, let's dive into the differences among FreeRTOS's dynamic-allocation schemes – the five heap-implementations.

FreeRTOS heap implementations

Because FreeRTOS targets such a wide range of MCUs and applications, FreeRTOS offers five different types of **heaps**. Each heap type provides a particular set of features for dynamic allocation. When FreeRTOS is built, one of the heap types is used. The heap types are implemented in the FreeRTOS program-files `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c`, and `heap_5.c`. These program files are distributed with the FreeRTOS source-code, and they are in the directory `Source/portable/MemMang`.



A note on memory pools:

Many other RTOSes include memory pools as an implementation for dynamic memory-allocation. A memory pool achieves dynamic allocation by only allocating and freeing fixed-size blocks. By constraining the block to a fixed-size, the problem of fragmentation is avoided in memory-constrained environments.

The downside to memory pools is that the blocks need to be sized for each specific application. If they are too large, they will waste precious RAM; if too small, they will be unable to hold large items. To make things easier on users and to avoid wasting RAM, Richard Barry elected to only use heaps for dynamic allocation in FreeRTOS.

In order for a FreeeRTOS project to properly compile and link, it is important to only have one of the heap program-files visible to the build-configuration. This can be accomplished by removing the unused heap program-files, or by not including the unused heap program-files in the list of files to be compiled.

For the prior chapters' example-programs that use FreeRTOS, `heap_4.c` is used. `heap_4.c` resides in the directory `Middleware\Third_Party\FreeRTOS\Source\portable\MemMang`. The four other heap program-files were removed from that directory. For those prior chapters' example-programs, their build-configurations compile and link whatever is in the directory `MemMang`.

For the present chapter's example-programs, their build-configurations use a specific heap program-file in the directory `Src/MemMang`. That directory has all five of the heap program-files. The build-configurations use `heap_1.c`. A build-configuration could be changed to use a different heap program file, if it is compatible with the example-program's use of the heap.

The following sections describe the five heap-types and how they can be used. To understand the heap types, we first need to know about the FreeRTOS APIs **pvPortMalloc()** and **vPortFree()**.

When the FreeRTOS kernel requires RAM it calls `pvPortMalloc()`, instead of calling `malloc()`. To free RAM, the kernel calls `vPortFree()`, instead of calling `free()`. `pvPortMalloc()` has the same function prototype as the standard C-library `malloc()`, and `vPortFree()` has the same function-prototype as the standard C-library `free()`. `pvPortMalloc()` and `vPortFree()` are public functions, so they can also be called from application code.

`pvPortMalloc()` and `vPortFree()` are defined in the `heap_*.c` program-files. Each heap implementation has its own version of those functions.

The five heap implementations

All of the various heap types exist to enable a project to get exactly the functionality it needs without requiring anything more (in terms of program-space or configuration). The heap types also allow for trade-offs between flexibility and deterministic timing. The following is a list of the five heap-types:

- `heap_1`: Allocation only – no freeing of memory is allowed. This is the simplest heap implementation, and it is best suited for applications that don't free any memory after initial creation. This implementation is deterministic (`allocation` always takes the same amount of time to execute). And, it's the only heap implementation that is deterministic. Since freeing of memory is not allowed, `heap_1` does not support `vPortFree`. Instead, it is implemented as a stub that puts the system in a spin loop, to identify its misuse. This is illustrated by the example-program `main_heap1_DeleteAttempt.c`.
- `heap_2`: Allocation and freeing are both allowed, but adjacent free blocks are not combined. This limits use-cases to applications that can reuse those freed items. This heap implementation is not a great fit for applications that call `vPortMalloc` and `vPortFree` `themselves`, unless there is a large degree of discipline in ensuring that only a small set of sizes are used. The implementation is not deterministic, but it is much more efficient than most standard C-library `malloc()` implementations.
- `heap_3`: Wraps standard `malloc` and `free` implementations to provide thread safety. It is not deterministic.
- `heap_4`: The same as `heap_2`, but it combines adjacent free space. Also, it allows relocating the entire heap to an absolute address. It is not deterministic, but it is much more efficient than most standard C-library `malloc()` implementations. It is well-suited for applications that use dynamic allocation. All of the book's FreeRTOS examples use heap_4 except those in the current chapter.
- `heap_5`: The same as `heap_4`, but it allows for creating a heap that is distributed across different noncontiguous memory regions – for example, a heap could be scattered across internal and external RAM.

Comparing FreeRTOS heap implementations

Here's a quick comparison between all of the heap implementations:

Heap implementation	Thread safe	Allocation	Free	Combine adjacent free-space	Multiple memory regions	Deterministic
`heap_1.c`	✓	✓				✓
`heap_2.c`	✓	✓	✓			
`heap_3.c`	✓	✓	✓	✓*		
`heap_4.c`	✓	✓	✓	✓		
`heap_5.c`	✓	✓	✓	✓	✓	
`std C lib`	?	✓	✓	✓*		

In the table, the heap-implementation "std C lib" is a standard C-library.

In the column "Combine adjacent free-space", two heap-implementations are tagged by "*". They are `heap_3.c` and a standard C-library. For these heap-implementations, most, if not all, will combine adjacent free-space.

The column "Deterministic" refers to whether the allocation functions always take the same amount of execution time, and the free functions always take the same amount of execution time. For `heap_3.c` and the standard C-library, the variations in execution time depend on the C-library implementation being used. Typically, general-purpose heap implementations are created to minimize fragmentation, as in a C-library. This requires additional CPU resources (time), and it increases the variation in the allocation and free functions' execution time (depending on how much memory is moved around).

Each C-library implementation may approach dynamic-allocation differently. For `heap_3.c`, some C-libraries will make adding thread-safety as easy as defining implementations for `__malloc_lock` and `__malloc_unlock`, in which case a single mutex is all that is required. Other cases can be more complex.

Choosing your FreeRTOS heap implementation

So, how do you go about choosing which heap implementation to use? First, you need to ensure that you're able to use dynamic-allocation (many standards for safety-critical applications disallow it). If you never need to free allocated memory, then `heap_1.c` is a potential option (as is avoiding a heap entirely).

From a coding perspective, the main difference between using `heap_1` and static-allocation is when the checks for memory availability are performed. When using the `*CreateStatic` variants, you'll be notified at link time that you don't have enough memory to support the newly created primitive. This requires a few extra lines of code each time a primitive is created (to allocate storage used by the primitive). When using `heap_1`, as long as checks are performed to determine task-creation success, then the checking will be performed at runtime (see *Chapter 7, Running the FreeRTOS Scheduler*). Many applications that are appropriate for the `heap_1` implementation will also create all required tasks before starting the scheduler. Using dynamic memory-allocation in this way isn't much different from static-allocation; it simply moves the checking from link time to runtime, while reducing the amount of code required to create each RTOS primitive.

When `heap_1` is used, `xTaskCreate()` can be used to create a task, but `vTaskDelete()` cannot be used to delete it. Storage for the task's stack and TCB will be obtained from the heap. However, `vTaskDelete()` cannot be called for the task, as that storage cannot be freed. Calling `vTaskDelete()` will cause the application to halt and go into an infinite spin-loop. This is shown by the example-program `main_heap1_DeleteAttempt.c`. The program's code-comments describe how to run the program, and how the program works. In general, when `heap_1` is used, the `*Create()` APIs can be used, but not their `*Delete()` counter-parts, e.g., `vQueueDelete`, `vSemaphoreDelete`, `vMutexDelete`, etc.

If you're working on an application that only requires *one* data type to be freed, `heap_2` might be an option. If you choose to go down this route, you'll need to be very careful to document this limitation for future maintainers of the code. Failure to understand the limited use-case of `heap_2` can easily result in memory fragmentation. In the worst-case scenario, fragmentation might potentially occur after the application has been running for an extended period of time, and it might not occur until the final code has been released and the hardware is fielded.

When dynamic memory is used, then `heap_3`, `heap_4`, or `heap_5` can be used. As mentioned earlier, `heap_3` simply wraps whatever C-runtime implementation of `malloc` and `free` is available, to make them thread safe. This way, they can be used by multiple tasks. This means that their behavior is going to be dependent on the underlying runtime implementation.

If your system has RAM in several different noncontiguous memory-locations (for example, internal and external RAM) then `heap_5` can be used to combine all of these locations into one heap. Otherwise, `heap_4` provides the same capabilities as `heap_5`, for allocation, freeing, and adjacent-block collation.

heap_4 and heap_5 are the two general-purpose heap implementations. Since they include code that will collate free blocks, it is possible that they will run for different periods of time when freeing memory. In general, it is best to avoid calls to vPortMalloc and vPortFree in code that requires a high degree of determinism. In heap_4 and heap_5, calls to vPortFree will have the most amount of timing variability, since this is when adjacent-block collation occurs.

In general, avoiding dynamic-allocation will help to provide more robust code with less effort – memory leaks and memory fragmentation are impossible if memory is never freed. On the other end of the spectrum, if your application makes use of standard library-functions, such as printf and string manipulation, you'll likely need to replace the library's versions of malloc and free with thread-safe implementations. Let's take a quick look at what's involved in making sure other parts of the application don't end up using a heap-implementation that isn't thread safe.

Replacing malloc and free

Many C-runtimes will ship with an implementation of malloc. However, the C-runtime versions for embedded-systems are not necessarily thread-safe by default. Because each C-runtime is different, the steps needed to make malloc thread safe will vary. The included STM toolchain used in this book includes **newlib-nano** as the C-runtime library. The following are a few notes regarding newlib-nano:

- newlib-nano is a derivative of newlib, targeted for embedded systems. It uses similar APIs as newlib and some of the information about newlib applies to newlib-nano.
- newlib-nano uses malloc and realloc for stdio.h functionality (for example, printf). (realloc is a library-function for resizing a memory-block that was previously obtained by a library-function such as malloc.)
- realloc is not directly supported by FreeRTOS heap implementations.
- FreeRTOSConfig.h includes the configUSE_NEWLIB_REENTRANT setting to make newlib thread safe, but it needs to be used in conjunction with the appropriate implementations of all stubs. This will allow you to use newlib-based printf, strtok, and so on in a thread-safe manner. This option also makes general calls to malloc and free safe to use from anywhere, without needing to explicitly use pvPortMalloc and vPortFree.



For more information on using newlib, see the section "*Further reading*". The referenced document describes how to use newlib safely in a FreeRTOS project, with the GNU toolchain.

Luckily, there aren't any calls to raw `malloc` in the example code included in this book. Normally, the STM HAL USB CDC implementation would include a call to `malloc`, but this was converted to a statically defined variable instead, which enables us to simply use the heap implementations included with FreeRTOS.



The `malloc` call in the STM-supplied USB stack was especially sinister because it occurred inside the USB interrupt, which makes it especially difficult to guarantee thread safety during `malloc`. This is because, for every call to `malloc`, interrupts would need to be disabled from within the tasks and also within interrupts that made calls to `malloc` (USB in this case). Rather than go through this trouble, the dynamic allocation was removed altogether.

Now that we've come to terms with different safety options using dynamic allocation, let's take a look at some additional tools that FreeRTOS has for reporting the health of our stacks and heaps.

Monitoring for stack and heap problems

When many people first start programming with an RTOS, one of the immediate challenges is figuring out how to properly size the stack for each task. This can lead to some frustration during development because when a stack is overrun, the symptoms can range from odd behavior to a full system crash.

Keeping an eye on stack space

FreeRTOS provides the option for using a stack-overflow callback function. The function is named `vApplicationStackOverflowHook`. If you choose to use it, it may save you from experiencing the oddball behavior or failure of an application that overruns its stack. When **configCHECK_FOR_STACK_OVERFLOW** is enabled in `FreeRTOSConfig.h`, any time a stack overflow is detected by FreeRTOS, the callback function will be run.

There are two potential values for `configCHECK_FOR_STACK_OVERFLOW`:

- `#define configCHECK_FOR_STACK_OVERFLOW 1`: Checks the stack-pointer location upon task exit.
- `#define configCHECK_FOR_STACK_OVERFLOW 2`: Fills the last 20 bytes of the stack with a known pattern and checks for the pattern upon exit.

The first method checks the task stack-pointer as the task exits the running state. If the stack pointer is pointing to an invalid location (where the stack shouldn't be), then an overflow has occurred:

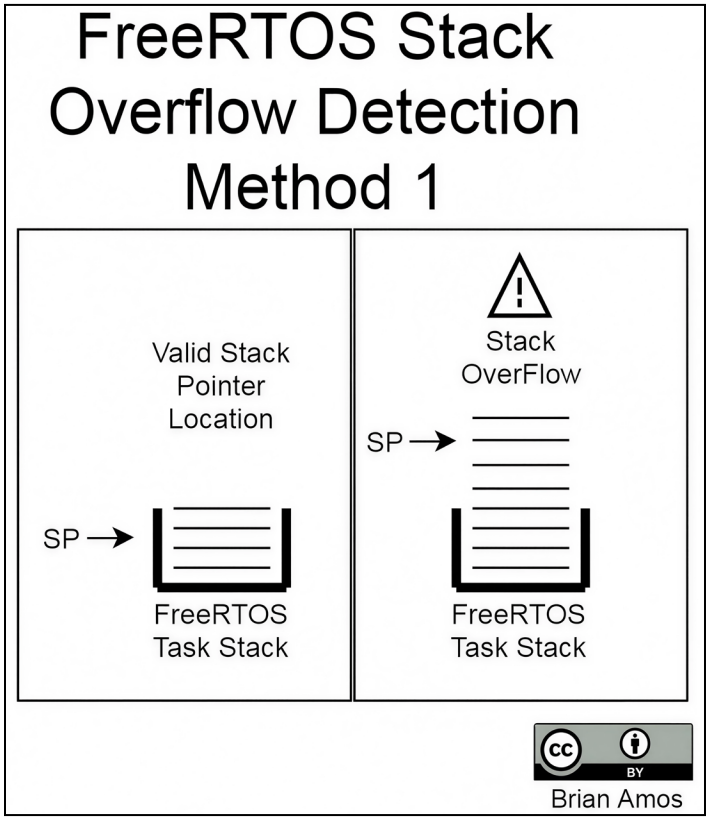


Figure 14.3: FreeRTOS stack overflow detection, method 1

This method is very fast, but it has the potential to miss some stack overflows. For example, if the stack has grown beyond its originally allocated space, but the stack pointer happens to be pointing to a valid spot when checked, then the overflow will be missed. To combat this scenario, a second overflow-detection method is provided.

To use the second method, set `configCHECK_FOR_STACK_OVERFLOW` to 2. It is an extension of method 1. Like method 1, it checks the location of the stack-pointer after the task exits the running state. In addition, it sets the last 20 bytes of stack with a known pattern. Upon task exit, it checks whether those 20 bytes have changed from their expected value. If they have changed, an overflow is signaled.

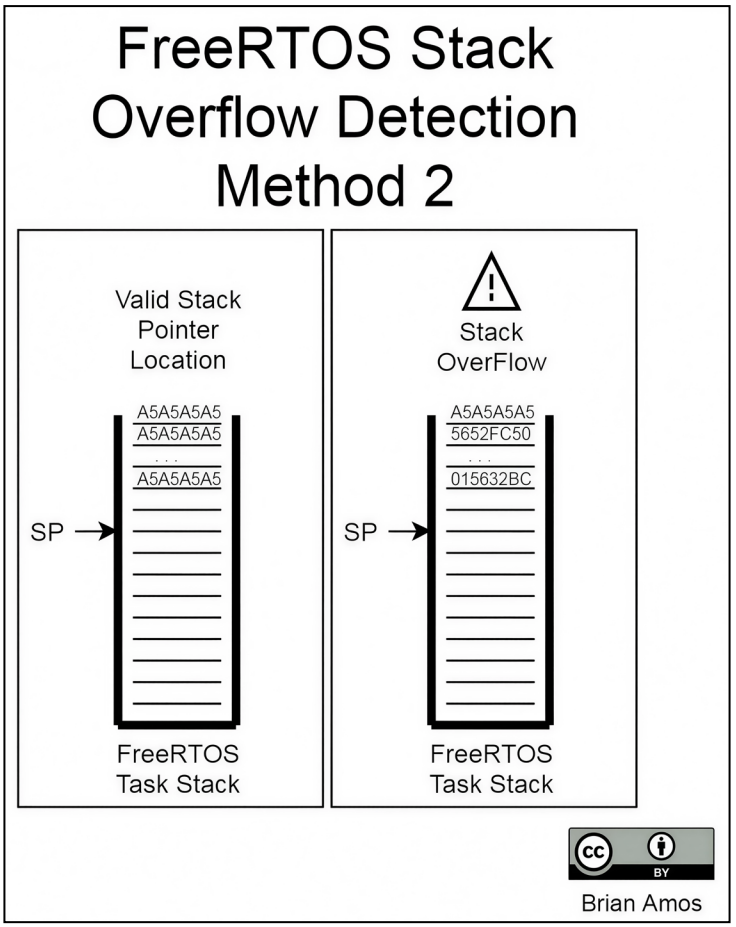


Figure 14.4: FreeRTOS stack-overflow detection, method 2

The trade-off of using this method is the extra time spent at task exit in checking the pattern. Also, the size of the stack needs to accommodate these extra 20 bytes.



While these methods are good for catching stack overflows, they are not perfect. For example, if an array is declared on a task stack and extends past the end of the stack, and only the end of the array is modified, then a stack-overflow won't be detected. There are other scenarios that can *fool* the overflow detection algorithm. But most of those scenarios are pretty contrived. In most cases, the stack-overflow detection-methods used by FreeRTOS will be sufficient to catch a task's undersized stack in your application.

To implement a very simple hook that will stop execution when a stack overflow occurs, we'll take the following simple steps:

1. In `FreeRTOSConfig.h`, define the configuration flag:

```
#define configCHECK_FOR_STACK_OVERFLOW 2
```

2. In a `*.c` file, add the stack-overflow hook:

```
void vApplicationStackOverflowHook( void )
{
    __disable_irq();
    while(1);
}
```

This very simple method disables all interrupts and executes an infinite loop, leaving no question that something has gone wrong. At this point, a debugger can be used to analyze which stack has overflowed.



This is probably not the method you would want to use in production-quality code. This simple method is good during development when many eyes are on the code, and a debugger is readily available. In production, you will want to have some sort of error trap or routine that places all of the hardware into a safe state, and possibly informs the user of the error. (Informing the user may not be possible in many embedded applications!)

Keeping an eye on heap space

If your application makes regular use of the FreeRTOS heap, then you should strongly consider using the **configUSE_MALLOC_FAILED_HOOK** configuration option, and the associated hook, `vApplicationMallocFailedHook`. This hook is called anytime a call to `pvMalloc()` fails.



Of course, while you're doing this, you're being a responsible programmer and checking the return value of `malloc`, and handling these error cases anyway... so this hook may be redundant.

The steps for setting this up are the same as in the previous hook:

1. Add the following in `FreeRTOSConfig.h`:

```
#define configUSE_MALLOC_FAILED_HOOK 1
```

2. In a `*.c` file, add the failed-`malloc` hook:

```
void vApplicationMallocFailedHook( void )
{
    __disable_irq();
    while(1);
}
```

There are also two helpful API functions that can be called on a regular basis to get a general sense of the available heap-space, **xPortGetFreeHeapSize** and **xPortGetMinimumEverFreeHeapSize**.

These functions return the available heap-space and the least amount of free heap-space ever recorded. They don't, however, give any clue as to whether or not the free space is fragmented into small blocks.

Another helpful API for monitoring heap-space is **vPortGetHeapStats**. It fills a struct with half-a-dozen values related to stack usage. The API is only available for `heap_4` and `heap_5`.

So, what happens if none of these safeguards provide enough peace of mind that each of your tasks is playing nicely with the rest of the system? Read on!

Using a memory protection unit (MPU)

A **memory protection unit (MPU)** continuously monitors memory access at a hardware level to make absolutely certain that only legal memory accesses are occurring. Otherwise, an interrupt is raised and immediate action can be taken. This allows many common errors to be immediately detected, which might otherwise go unnoticed for a period of time.

Problems like stack-overflows can be immediately caught when using an MPU, e.g., a stack growing into the memory space reserved for another task. Buffer overflows and pointer errors are also stopped dead in their tracks when an MPU is utilized, which makes for a more robust application.

The STM32F767 MCU includes an MPU. In order to make use of the MPU in FreeRTOS, the MPU-enabled FreeRTOS port must be used: `GCC\ARM_CM4_MPU`. (The book's example-programs do not use this port.) This way, restricted tasks can be created by using the API `xTaskCreateRestricted`, which contains the following additional parameters:

```
typedef struct xTASK_PARAMTERS
{
    pdTASK_CODE pvTaskCode;
    const signed char * const pcName;
    unsigned short usStackDepth;
    void *pvParameters;
    unsigned portBASE_TYPE uxPriority;
    portSTACK_TYPE *puxStackBuffer;
    xMemoryRegion xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} xTaskParameters;
```

Restricted tasks have limited execution rights and limited memory-access rights.

`xTaskCreate` can be used to create tasks that either operate as standard user-mode tasks or privileged-mode tasks. In privileged mode, a task has access to the entire memory map, whereas in user mode, it only has access to its own flash and RAM, which isn't configured for privileged-access only.

In order for all of this to come together, the MPU ports of FreeRTOS also require variables to be defined in the linker file. (These variables will be placed into the `*.LD` file.)

Variable name	Description
`__FLASH_segment_start__`	The start address of the flash memory
`__FLASH_segment_end__`	The end address of the flash memory
`__privileged_functions_end__`	The end address of the `privileged_functions` named section
`_SRAM_segment_start__`	The start address of the SRAM memory
`__SRAM_segment_end__`	The end address of the SRAM memory
`__privileged_data_start__`	The start address of the `privileged_data` section
`__privileged_data_end__`	The end address of the `privileged_data` section

Additional information on FreeRTOS's MPU-support can be found in the section "*Further reading*".

Summary

In this chapter, we've covered a) static and dynamic memory-allocation, b) all of the available heap implementations in FreeRTOS, and c) how to implement hooks that let us keep an eye on our stacks and heaps. By understanding the trade-offs to be made when using the different allocation schemes, you'll be in a good position to choose an appropriate method for your ` future projects.

In the next chapter, we'll discuss some of the details of using FreeRTOS in a multicore environment.

Further reading

- *The Power of 10: Rules for Developing Safety-Critical Code*, by Gerard Holzmann `http://web.eecs.umich.edu/~imarkov/10rules.pdf`
- An article on newlib and on FreeRTOS re-entry, by Dave Nadler `http://www.nadler.com/embedded/newlibAndFreeRTOS.html`

Sources on memory management from FreeRTOS.org and ST are listed in *Appendix B*, "*Reference Information*". The sources include tutorials, an example-program, user manuals, and reference manuals.

15

Multi-Processor and Multi-Core Systems

So far, we've discussed many different ways of programming a single **microcontroller unit (MCU)**. But what if the task at hand requires more processing than a single-core MCU can supply? What if the mechanical constraints of the system dictate the use of multiple MCUs physically distributed in the system while working together to complete a task? What about cases where reliability is paramount and a single failed processor results in a catastrophic system failure? All of these cases require the use of more than one processing core and, in some cases, more than one MCU.

This chapter explores multi-core and multi-processor solutions and their different applications. First, we'll take a look at the different design requirements that might drive a multi-core/processor solution. We'll then dive a bit deeper into the different ways FreeRTOS can be used in multi-core/processor systems. Finally, some recommendations on choosing an inter-processor communication scheme will be presented.

In a nutshell, we will cover the following topics:

- Introducing multi-core and multi-processor systems
- Exploring multi-core systems
- Exploring multi-processor systems
- Exploring inter-processor communication
- Choosing between multi-core and multi-processor systems

Introducing multi-core and multi-processor systems

First, let's get our terminology straight. A **multi-core** design is a single chip with multiple CPUs inside it, with at least some memory shared between the cores:

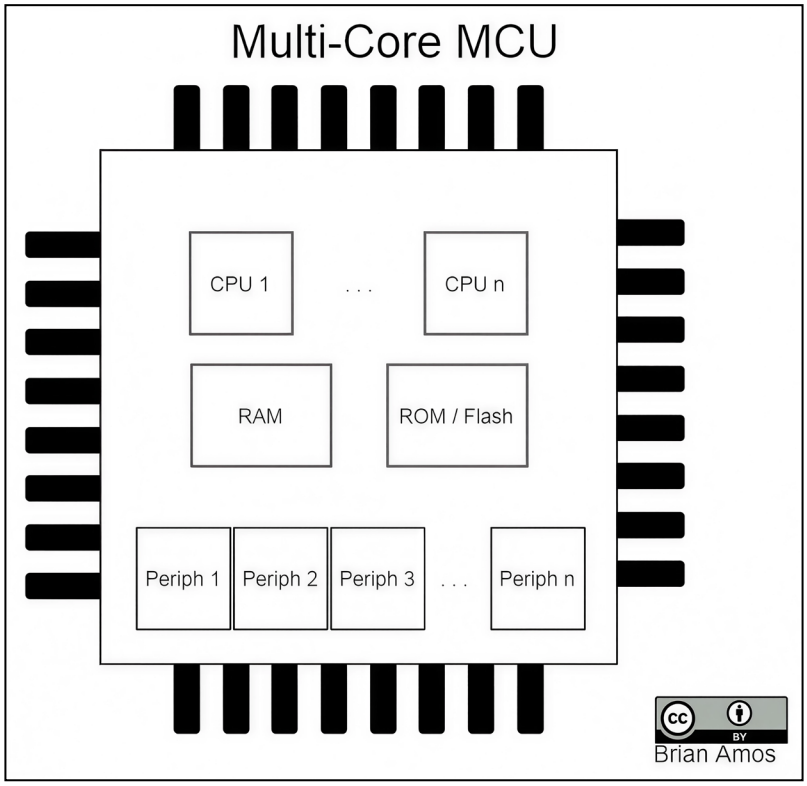


Figure 15.1: Block diagram of a multi-core MCU

Multi-core systems span a very broad range, from the larger 64-bit systems that have multiple identical CPU cores, to the ARM big.LITTLE architecture, which incorporates both high-bandwidth CPUs and power-conscious MCUs in the same package. Recently, multi-core MCUs have also become more commonly available. **Graphics processing units (GPUs)** can also be grouped into the multi-core category.

A **multi-processor system** is one where there are multiple processor chips in the design. For the purposes of our discussions here, these chips can reside on the same **printed circuit board assembly (PCBA)** or different PCBAs distributed throughout a larger system:

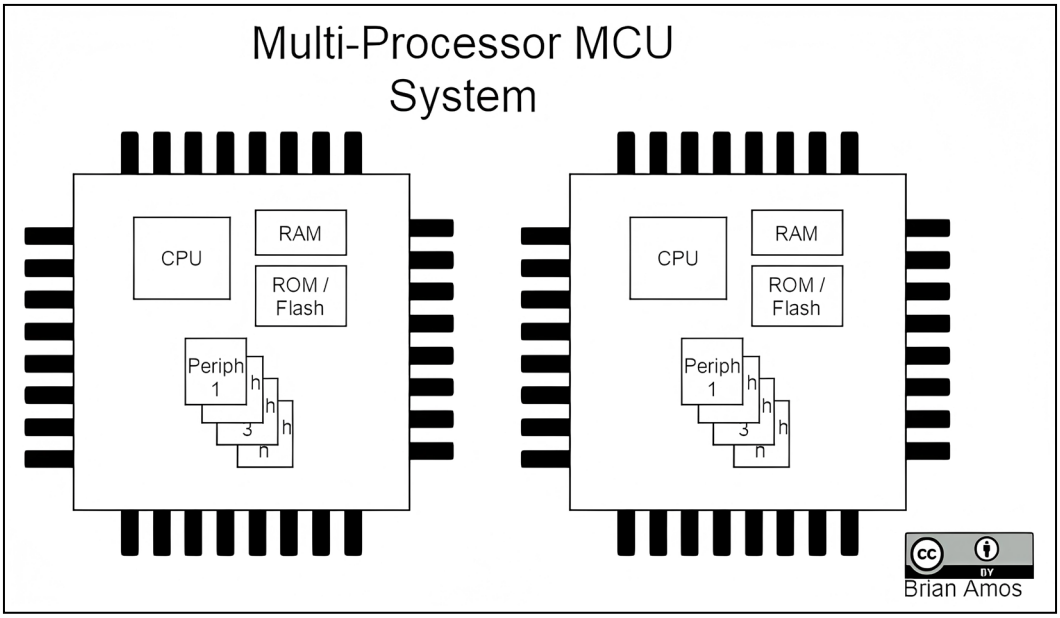


Figure 15.2: Block diagram of a multi-processor MCU system

Both multi-core and multi-processor topologies can be found in many different places, such as smartphones, small networked sensing devices, industrial automation equipment, test equipment, medical equipment, appliances, and of course, a range of computing devices, such as desktops, laptops, and so on.

There are many different driving forces for using these two different topologies, beyond a simple need for more or faster processing. Sometimes, a system needs to come online *instantly*, without waiting for a full **general-purpose operating system (GPOS)** to boot. Occasionally, there are regulatory requirements that are easier to meet by segregating system functions into multiple cores (and code bases) so that only a portion of the total code (or system) is required to go through a stringent review. There could be electro-mechanical considerations in a system (such as long wire-runs to motors/actuators, or sensitive analog-signals) that are best addressed by having a processor in close physical proximity. In high-reliability systems, redundancy is very common.

Now that we have a general idea of the terminology, let's get into some additional details and use-cases for these systems, starting with multi-core designs.

Exploring multi-core systems

First, let's cover a few different types of multi-core systems. They have two primary types of configurations/architectures: heterogeneous and homogeneous. A heterogeneous system is one that has multiple cores, but they are different in some way. Contrast this with a homogeneous system, where all CPUs can be treated identically and interchangeably.

Heterogeneous multi-core systems

A heterogeneous multi-core system has at least two processing cores in the same device and includes differences in either the processor-architecture of the core, or the way the cores access shared resources, such as system memory, peripherals, or I/O. For example, at the lower end of the spectrum, we can have multiple MCU cores on the same chip. The LPC54100 series from NXP incorporates a Cortex-M0+ and a Cortex-M4, both running at 150 Mhz, in the same package.

In this device, the MCU cores are different, but their connection to system-peripherals is identical – except for instruction and data buses, which are only available on the Cortex-M4:

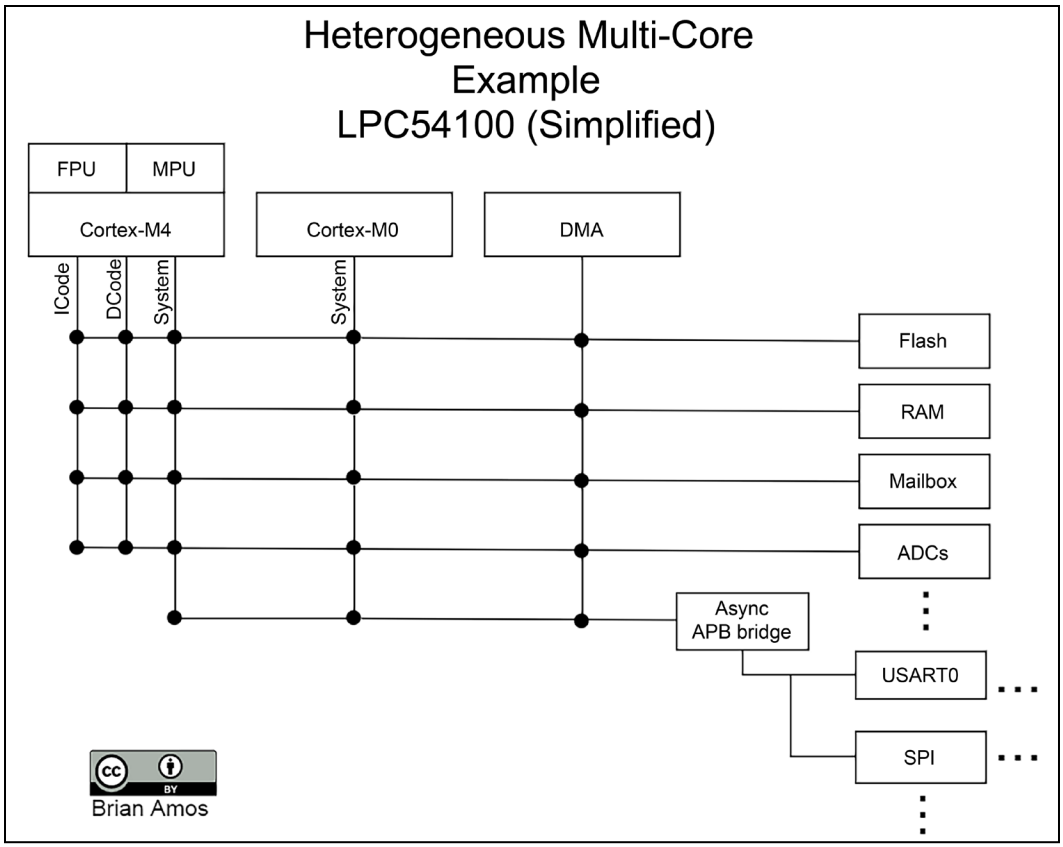


Figure 15.3: Block diagram of a heterogeneous multi-core MCU (NXP LPC54100)

We can use systems like these in different ways:

- **Segmenting hard real-time operations from more general-purpose computing**: the M0+ could handle low-level peripheral or hardware access/control, while the M4 handles the higher-level functionality required, such as GUIs and connectivity.
- **Power conscious design**: Low-level control and interfacing is performed on the lower-power M0+, only activating the M4 when computationally-expensive operations are required.

Since the LPC54100 has two MCU cores, we'll focus on the use of: a) bare-metal programming (no operating system), and b) operating systems that don't require a full-blown **memory management unit (MMU)**, such as FreeRTOS. Running different (or multiple copies of the same) operating systems on the two cores is called **asymmetric multi-processing**.



The name asymmetric comes from the fact that the two cores are treated differently from one another – there is *asymmetry* between them. This is quite a bit different from the symmetric multi-core approach that is used on desktop-based operating systems, where the various cores are all treated equally. Symmetric multi-core systems will be covered in the *Homogeneous multi-core systems* section.

For example, we could run a copy of FreeRTOS on each of the two cores:

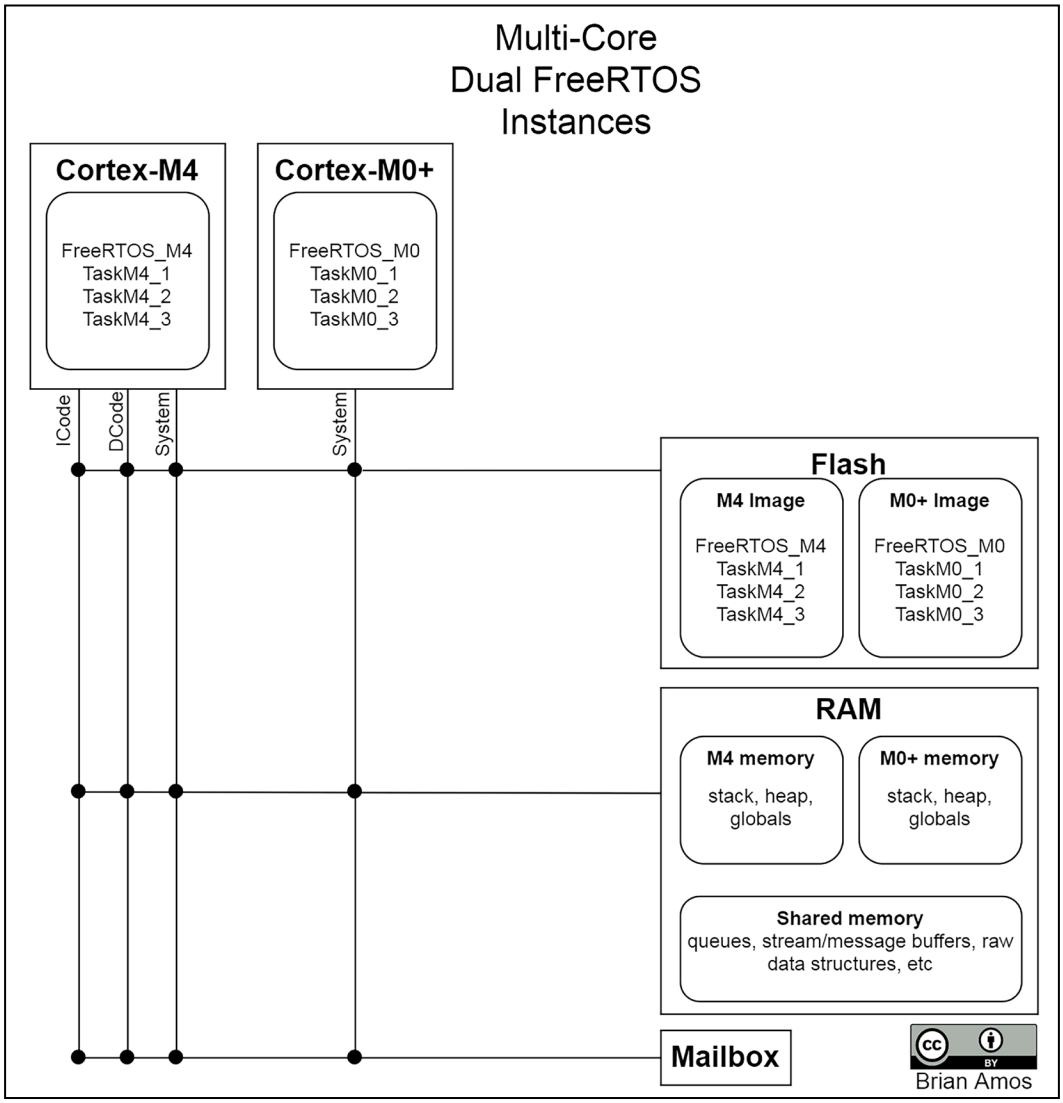


Figure 15.4: Multi-core MCU with dual FreeRTOS instances

In a configuration like this, the two cores run completely independently from one another. Even though FreeRTOS is being run on both cores, there is no flash program-space shared between the cores – each core has a firmware image that is *independent* from the other. RAM behaves in the same way – the full RAM memory-space is available to both cores, but by default, each core will receive its own area for stack, heap, global variables, and so on.

So, each core is running its own program – how do the two programs coordinate activities between each other? We need some way of passing information back and forth – but how?

Inter-core communication

Information sharing between the cores is possible, but it is subject to the same concurrent-access considerations that any other multi-threaded environment has. For this reason, mailbox hardware is typically included on the device. This hardware is dedicated to facilitating communication between the two cores. Mailboxes will generally have the following features:

- **Hardware mutex functionality**: Used to protect RAM shared between the two cores. The idea is identical to mutexes in a pure software environment – they are used to provide mutually exclusive access to a shared resource.
- **Interrupts to/from each core**: These interrupts can be raised by a core after writing data to a shared area of memory, alerting the other core that a message/data is available.

Legacy application extension

We're not limited to running FreeRTOS on both cores – any mixture of RTOSes or bare metal can be mixed or matched between the cores. Let's say a bare-metal legacy application already existed but some new additional functionality was required to take advantage of a new opportunity. For example, to stay competitive, the device might need a *facelift* and have a GUI, web frontend, or IoT stack added to it. The new functionality could potentially be developed separately from the underlying legacy code, leaving the legacy code largely intact and undisturbed.

For example, the legacy code could be run on the Cortex-M0+, while the new functionality is added to the Cortex-M4:

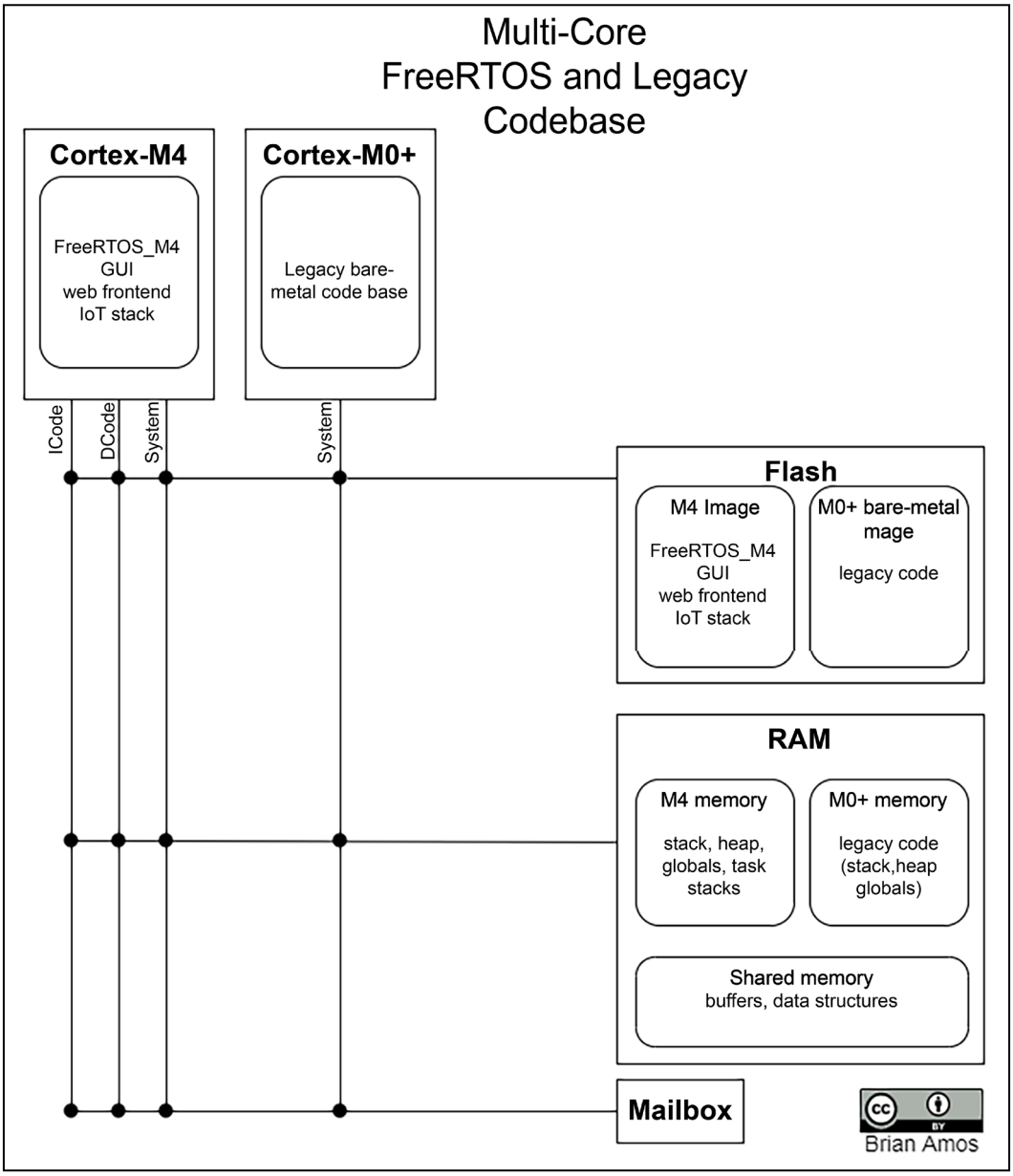


Figure 15.5: Multi-core system using a legacy codebase on one CPU, and FreeRTOS on the other

In a design like this, whether shared RAM is used as a data exchange between the cores will depend greatly on how comfortable a team is in modifying the legacy code-base and how the application is structured. For example, rather than modifying an existing codebase to use proper mailbox-implemented mutexes before accessing a shared data-structure, it might be preferable to use a pre-existing hardware interface as the data-transfer mechanism, treating the secondary CPU more like an external client. Since many legacy systems use UARTs as the primary interface to the system, it is possible to use these data streams as an interface between the processors, keeping modifications to the legacy code to a minimum:

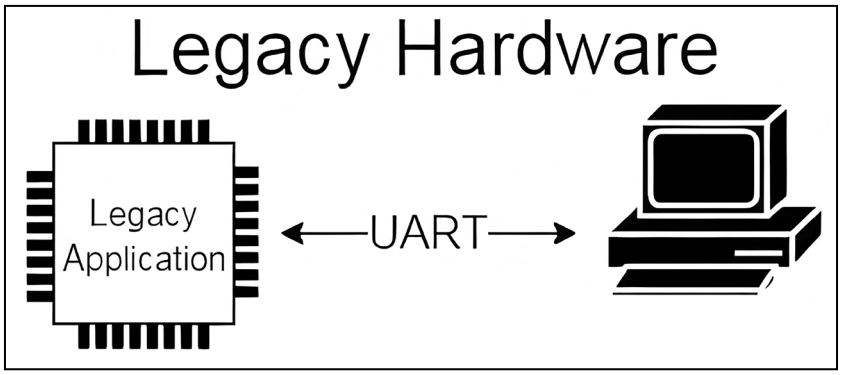


Figure 15.6: Legacy application using a UART to communicate with a PC

This approach avoids significant modifications to the legacy code-base at the expense of a) using a slower interface (physical peripherals are slower and more CPU-intensive than simple memory transfers) and b) routing signals outside the processor. Although far from ideal, this approach can be used to test the viability of a new opportunity before investing significant engineering effort in a more elegant solution:

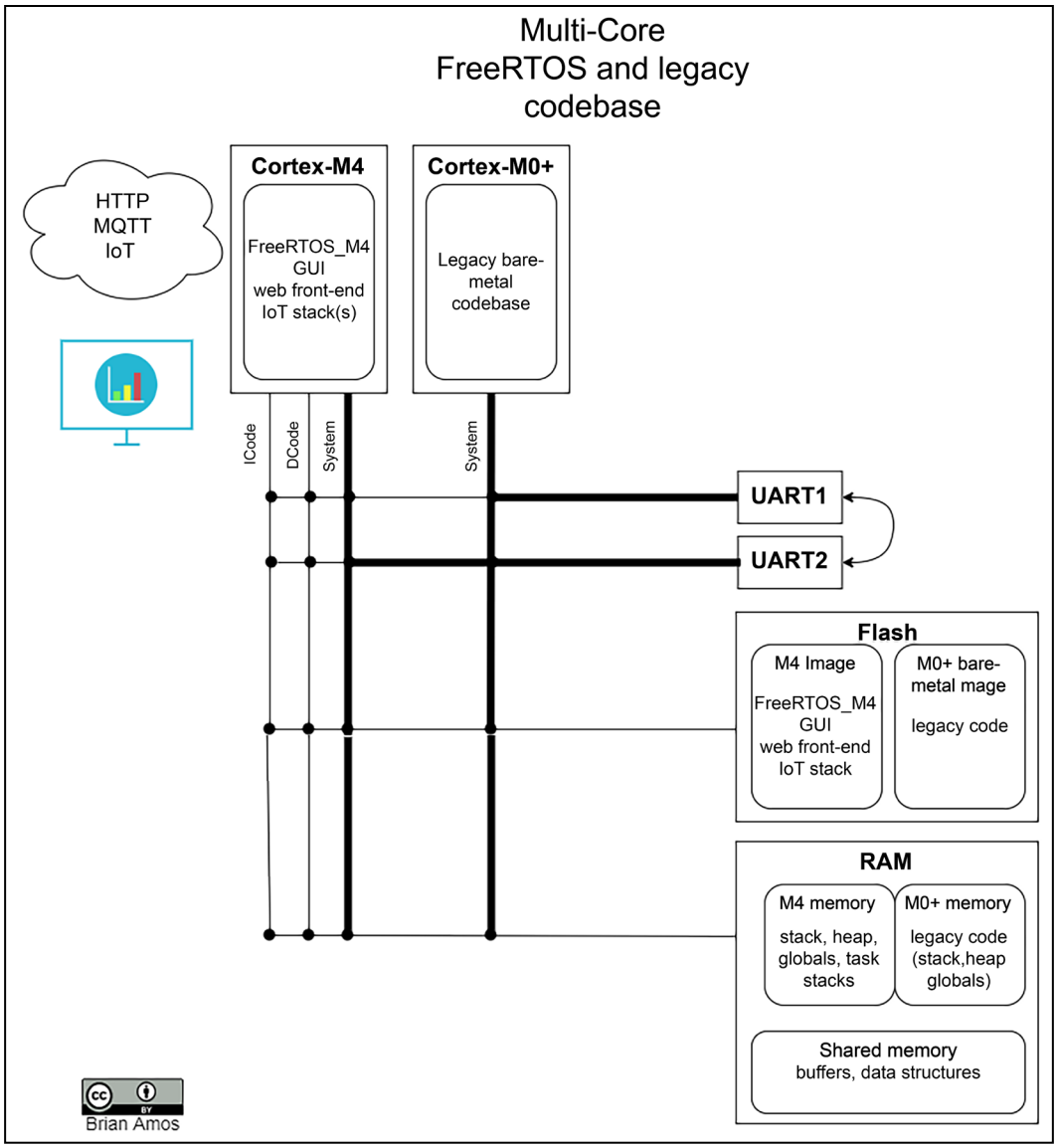


Figure 15.7: Example of legacy code communicating with a new FreeRTOS application, using UARTs

This type of approach allows the team to focus on developing new interfaces for an existing system—whose core functionality doesn't need to change—with minimal impact on the original system.



Depending on the circumstances, it may also make more sense to leave the legacy code on the original MCU, rather than porting it to a core inside a new MCU. Each project will likely have its own constraints required to guide this decision. Although all of this might look simple from a very high level, each project usually has some hidden complexities that need to be considered.

High-demand hard real-time systems

We just looked at the NXP LPC54100, which is at the lower end of the spectrum of heterogenous multi-core systems. At the other end of the spectrum would be a device such as the NXP i.Mx8, which contains two Cortex-A72s, four Cortex-A53s, two Cortex-M4Fs, one DSP, and two GPUs. Such a system will generally be used where computationally-intensive operations are required, in addition to low-latency, or hard real-time interactions with hardware. Reasonable applications for the i.Mx8 include computer vision, AI, on-target adaptive machine-learning, and advanced closed-loop control-systems.

However, instead of incorporating an i.Mx8 (or similar CPU) into a product, why not use a more general-purpose computing solution for a system that requires this much computing power? After all, general-purpose computers have had GPUs and multi-core CPUs for a decade or more, right?

In some systems, it might be perfectly acceptable to run general-purpose computing-hardware, with a general-purpose operating-system (GPOS). However, when there are **hard real-time requirements**, the system is considered to have failed if a real-time deadline was missed. And, in that case, a GPOS won't be sufficient. A compelling reason for using a device like the i.Mx8 is that they use cores such as the Cortex-M4, which are real-time-capable and have low latency. They are used to handle hard real-time tasks, where extreme reliability and low latency are paramount. Higher-throughput general-purpose hardware could be used in addition, for computational *heavy lifting*, in which higher latency and less determinism can be tolerated:

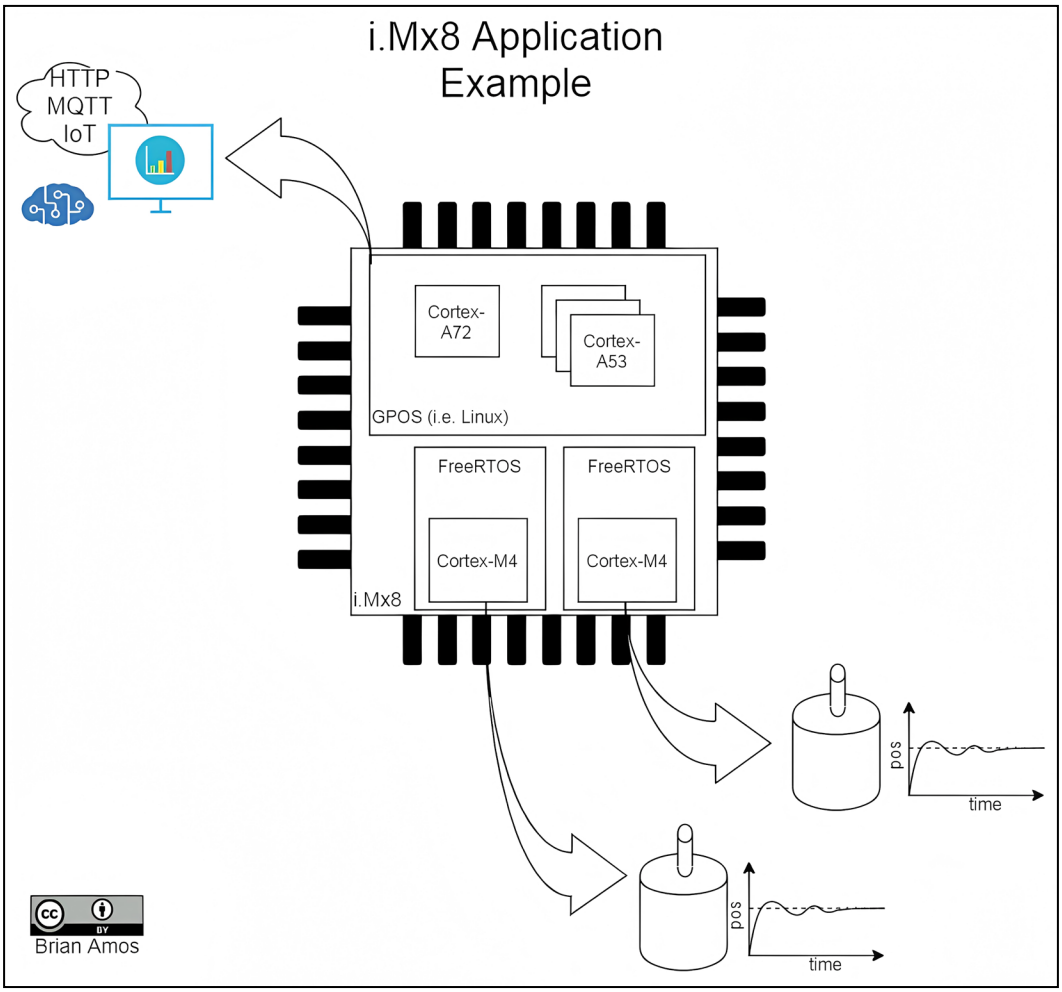


Figure 15.8: Application example using an NXP i.Mx8

The smaller MCU-based cores are extremely good at performing low-level exchanges with hardware such as sensors and actuators. Timing-sensitive operations requiring the use of specialized timing-peripherals are best left to the MCU hardware. For example, a motor-control system might require directly controlling an H bridge and reading data from an encoder that uses an obscure/proprietary timing format. This is fairly straightforward to implement using an MCU that has dedicated timing hardware. Differential PWM signals with dead-time insertion used for motor-control and high-resolution timing capture are both fairly common features. All of this tightly controlled and low-latency control structure can be implemented using the MCU and its specialized peripherals (either on bare metal or on an RTOS). And, higher-level commands can be exposed to a GPOS. Specifically, on the i.Mx8, we can now perform very low-level, timing-sensitive operations using MCUs. And, we can simultaneously perform high-level massively-parallel operations required for computer vision, machine learning, and AI. The latter are made possible by the higher-performance Cortex-A processors, DSP, and GPUs.



Heterogeneous systems aren't limited to embedded systems! Heterogeneous topologies have existed for very large computing clusters for decades, but we're keeping our focus on examples most relevant to the embedded space.

So, now that we've covered some examples of heterogenous multi-core systems, what about homogeneous multi-core systems?

Homogeneous multi-core systems

As you might expect from the name, a homogeneous multi-core system is one where all of the cores are the same. These types of multi-core systems have traditionally been found in desktop computing. Rather than having individual cores tailored to perform a few types of tasks very well (as with heterogenous systems), there are multiple cores that are all identical. Rather than programming individual cores with specific tasks that are tied to the cores, all of the cores are treated identically. This type of approach is referred to as **symmetric multi-processing**. (There is *symmetry* between all of the cores in the system.) In a symmetric system, cores will be exposed to a single kernel, rather than having multiple kernels/schedulers.

Even in asymmetric multi-processing setups, there can be components that are symmetric. For example, the i.Mx8 mentioned earlier will usually have the Cortex-A53 cores setup in a symmetric multi-processing arrangement. There, all four cores are available for scheduling by a single kernel, and all of the cores are treated identically by the kernel.

But what about when there is a need for processors in different physical locations? Or what if a single processor is limited in its functionality by the number of pins it has available? The next section describes how multi-processor systems can be used to solve such problems.

Exploring multi-processor systems

Multi-core systems are excellent for segmenting firmware functionality and providing parallel execution. Similarly, multi-processor systems are useful in many situations, for a variety of reasons. Let's take a look at a few examples.

Distributed systems

Embedded systems often have a very large amount of interaction with the physical world. Unlike the digital realm, where 1s and 0s can literally be sent around the world without a second thought, the physical world is a harsh place for sensitive analog signals – minimizing the distance traversed can be critical. It is a good idea to keep analog processing as close to its source as possible. For a mixed signal system with analog components, this means keeping the signal paths as short as possible and getting the sensitive analog signals processed and converted into their digital representations as close to the source as possible:

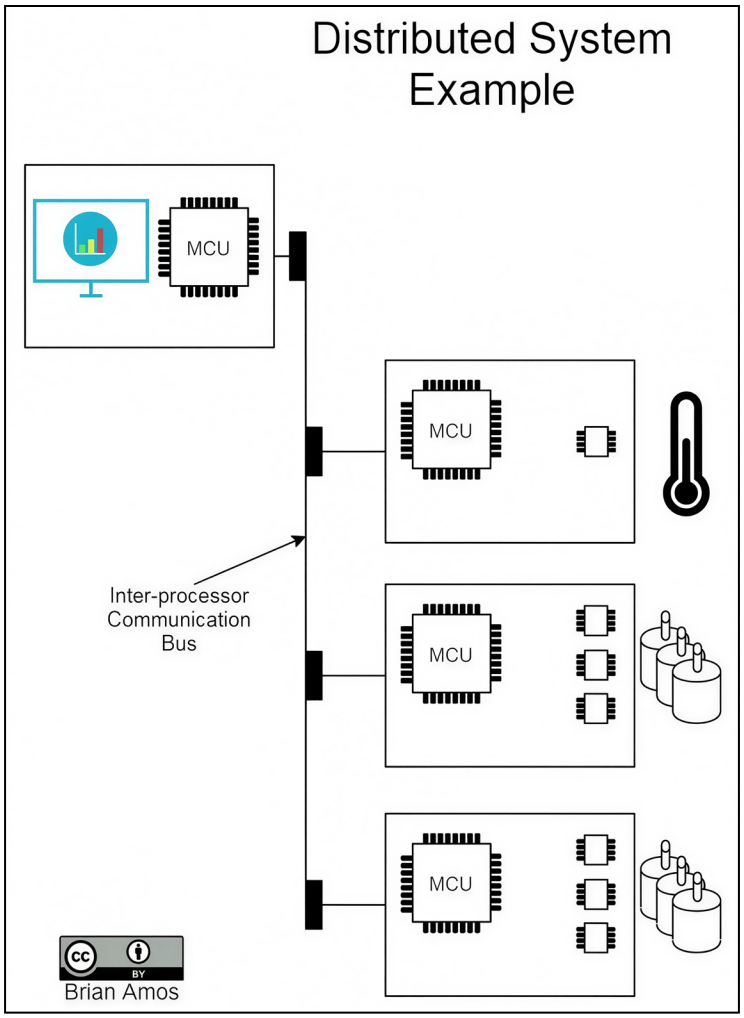


Figure 15.9: Example of a distributed system

In medium- to high-power systems, when wires carry current to control motors, solenoids, and other actuators, there are benefits in reducing the distance traversed by the wires. For example, it reduces the system's radiated electromagnetic-emissions. If the I/O in question is physically removed from the rest of the system, placing an MCU near a controlled device is a good way to localize digitization of the sensitive signals. It makes the system more immune to **electromagnetic interference (EMI)** while simultaneously minimizing the amount of wiring. Also, in high-vibration and high-motion environments, fewer wires mean fewer potential points of mechanical failure, and that results in higher reliability, less downtime, and fewer service calls.

Parallel development

Using multiple processors also makes it very easy to provide a level of parallelism in the actual development of the system. Since teams will often find it easiest to focus on a well-defined subsystem, creating multiple subsystems makes running true parallel development a possibility. Each subsystem can be demarcated by its own processor and communication interface, along with a clear list of the responsibilities of the subsystem:

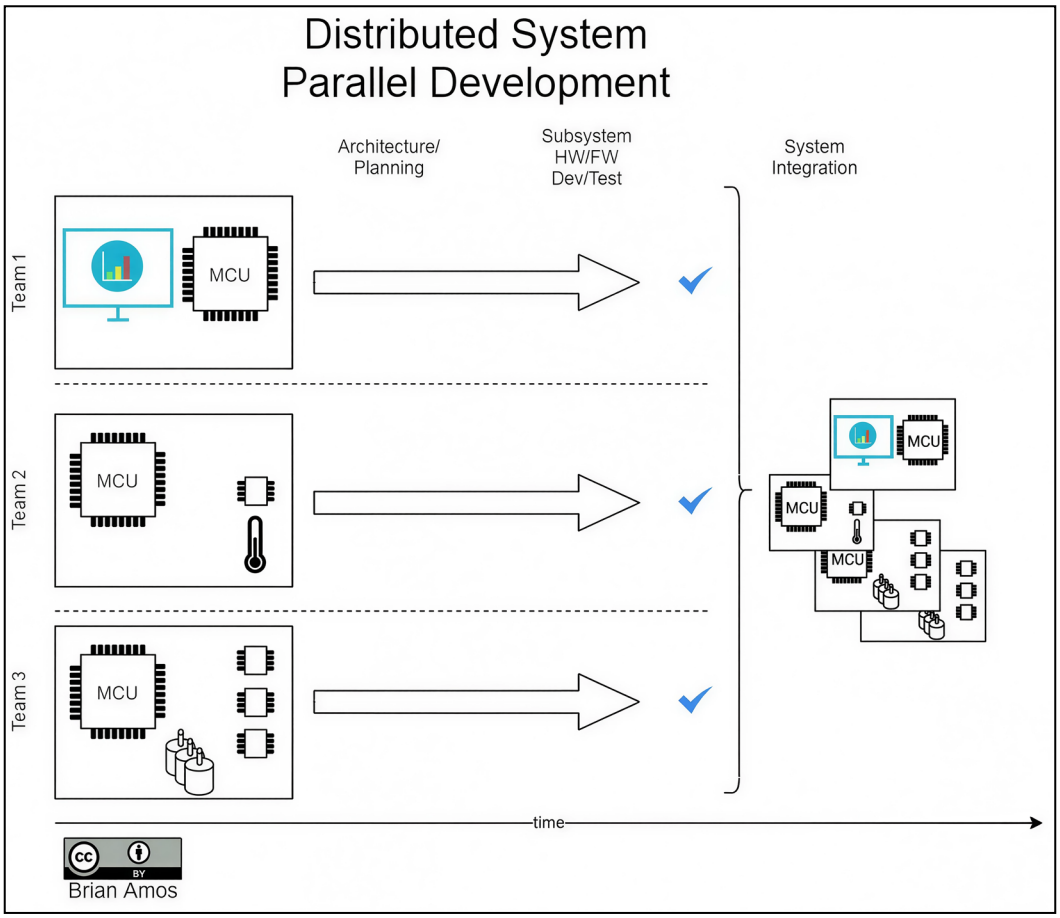


Figure 15.10: A distributed system is amenable to parallel development

This approach also has the advantage of encouraging each team to fully test their system in isolation, and to document the interfaces and functionality as they move through development. Finally, this approach tends to reduce surprises during integration, since the team is forced to put more thought into the entire architecture before starting development.

Design reuse

As processors begin to have a plethora of I/O connected to them, they may still have plenty of processing resources available but run out of available pins. At this point, there is a decision to make. There are ICs available that provide port expansion, but should they be used? If you're designing a system with reuse in mind, it is important to see whether a subsystem approach can be employed, instead of creating a huge monolithic design, where all of the hardware and firmware is intertwined and tightly coupled. Sometimes, when the pin capacities of a single MCU are reached, it is an indication that the MCU is performing the functionality of several different subsystems. Often, if these subsystems are broken down and individually developed, they can be *dropped* into future products without modification, which can greatly decrease future projects' risks and schedule.

High-reliability systems

High-reliability systems will often include multiple cores or processors for their critical functionality. However, rather than using this extra processing power to run individual parallel operations, they are set up for some level of redundancy. There are different ways to achieve redundancy. One path to creating a redundant system is for the cores to run in lockstep with one another. The results of each processor are meticulously checked against one another to detect any discrepancies. If a problem is found, the core (or processor) is taken off-line and reset, with a set of tests run to ensure it comes back up correctly – then, it is put back into service.

In systems like these, there can be environmental considerations, such as EMI from running motors, solenoid valves, or other actuators. Sometimes the source of the environmental noise is more extraordinary, such as solar radiation, which is often a concern for high-altitude and space-bound systems.

Now that we've explored the reasons why having multiple processors in a system can be useful, let's take a look at how to get all of these processors talking to one another.

Exploring inter-processor communication

Inter-processor communication was mentioned briefly in the context of distributed systems. Let's take a look at some of the considerations that go into choosing a suitable inter-processor bus.

Considerations for choosing the right communication medium

There are many considerations when choosing the communication medium used between processors. We will consider the following: timing, physical requirements, and development-team/project constraints.

Timing

In a real-time system, timing considerations are often some of the most important. If a message sent between nodes doesn't make it to its destination on-time and intact, it can have serious consequences. The following is a list of items to consider in regard to timing when trying to choose the right communication medium:

- **Latency**: How long will it take for a message to be sent and a response to be received? Having the ability to react quickly to communication between subsystems is often quite important.
- **Maximum jitter**: How much variability is there in the latency? Each system has its own requirements for how much variability is acceptable.
- **Error detection/reliability**: Does the communication-medium provide a way of determining whether a message was received correctly and on time?
- **Throughput**: How much data can be pushed over the communication medium? For communication mediums that contain control-data, throughput will often be measured in messages, rather than raw data (such as KB/sec or MB/sec). Often, maximum reliability and minimal latency will come at the cost of raw data-transfer throughput – each message will contain additional overhead and handshaking.

Physical Requirements

Sometimes, physical requirements are quite important, other times they may hardly be a constraint. Here are some physical requirements to consider when trying to choose the right communication medium:

- **Noise immunity**: Does the communication-channel need to run through an electrically noisy environment? What types of cabling are required for proper EMI shielding?
- **Number of nodes in the system**: How many nodes are required in the complete system? Most standards will have an upper bound on the number of connections due to electrical constraints.
- **Distance**: How long will the signal-distance between devices need to be? Will it be a short, chip-to-chip run within the PCB or a long run between buildings? Distributed systems can have widely different meanings to different developers and industries.
- **Required peripherals**: How much extra circuitry is acceptable? What kinds/sizes of connectors can be tolerated?

Development Team/Project Constraints

Sometimes the capabilities of the development team, or the constraints of the project, need to be considered when trying to choose the right communication medium:

- **Complexity**: How much code is required to get the protocol up and running? Has the required external circuitry been proven to be functional? Does our team feel like the features provided by the solution are worth the development time required to implement it?
- **Existing familiarity**: Has anyone on the team used this communication scheme before and is that experience directly relevant to the current project/product? Do we need to learn something new that is a better fit, rather than using something we're already comfortable with, but isn't the best solution?
- **Budget**: Does the communication scheme require any expensive components, such as exotic ICs, connectors, or proprietary stacks? Do we need to purchase part of the solution or use contractors to implement some of the solution?

As you can imagine from the long list of considerations, there is no *one-size-fits-all* communication mechanism that is an excellent fit for all applications. That's why we have so many possible solutions to choose from.

For example, while an industrial Ethernet communication-solution may provide excellent latency and noise performance, the fact that it requires specialized hardware can make it unsuitable for applications that do not need those features. On the flip side, a low-performance serial-protocol such as RS-232 may be extremely easy to implement but have an unacceptably high amount of EMI and be susceptible to noise when used at high speeds. On the other hand, the complexity of a full TCP/IP stack might put off many would-be adopters, unless someone on the team already has familiarity with it and a driver stack is readily available for the target platform.



In the end, this is what most professional developers grapple with: understanding trade-offs among competing solutions and employing those solutions that result in a profitable and useful product or service to their end-customers. It is often tempting to choose solutions that are either interesting or fun, rather than the one that is best for the application at hand.

Communication standards

From the previous list of considerations, we can see that choosing a method for inter-processor communication isn't one-size-fits-all. To help provide an idea of what solutions are available, here are some examples of commonly used buses for MCU-based systems and some brief commentary on how they might be useful in a multi-processor system. This list is by no means exhaustive. Also, each standard has its own merits under different circumstances.

Controller area network

A **controller area network** (**CAN**) is the communication backbone for many subsystems in the automotive industry. The advantages of CAN are its robust physical layer, a prioritized messaging scheme, and multi-master bus arbitration. Many MCUs include dedicated CAN peripherals, which helps to ease the implementation. CAN is most naturally suited for shorter messages, since the data field of extended frames may only contain up to 8 bytes.

Ethernet

Nearly all medium- to high-performance MCUs have provisions for **Ethernet**, requiring an external PHY, magnetics, and a connector for the hardware implementation. The challenge with Ethernet is ensuring suitable networking protocol stacks are available. Fortunately, there is a wide range of options for popular protocols that run on top of TCP and UDP, as well as readily available, inexpensive hardware that can be used to design a full network if required.

Similar to Modbus (described below), Ethernet will often be chosen as the externally facing interface, rather than an inter-processor bus. Depending on the system-architecture and hardware-availability, there might be good reason to use it for both.

Inter-integrated communication bus

Inter-integrated communication bus (**I2C**) is most often used for communicating with low-bandwidth peripherals, such as sensors and EEPROMs. Most often, an MCU will be configured as the I2C bus master with one or more ICs being I2C slave devices. However, many MCUs contain I2C controllers that can be used to implement either the master or slave side of I2C. Therefore, it is possible to use I2C as an inter-processor communication bus.

There are many aspects of the I2C protocol that make it non-deterministic, such as the ability for slaves to hold the clock line until they are ready to receive more data (clock stretching) and multi-master arbitration.

Local interconnect network

Local interconnect network (LIN) is a commonly used automotive network subsystem for a maximum of 16 nodes when a full CAN is too complex or expensive to implement. The LIN physical layer is less fault-tolerant than CAN, but it is also more deterministic, since there can only be one bus master. STM32 USARTS will often have some helpful LIN-mode functionality built into the peripheral, but an external PHY IC is still required.

Modbus

Modbus is a protocol that historically ran on top of an RS-485 physical layer and is very popular in the industrial space as an externally facing protocol (although these days, the protocol is commonly found on top of TCP). Modbus is a fairly simple register-oriented protocol.

Serial peripheral interface

A **serial peripheral interface (SPI)** can also be very useful as an easy-to-implement, highly deterministic inter-processor communication medium, especially when the accuracy of a slave isn't high enough to achieve the tight tolerances required for high baud-rates on an asynchronous serial port. All the same drawbacks for custom asynchronous protocols also exist for SPI-based custom protocols, with the additional constraint that slave devices will have hard real-time constraints imposed based on how quickly the master needs responses back from the slave(s).

Since the SPI clock is driven by the master, it is the only device that can initiate a transfer. For example, if a slave is required to have a response ready within 30 µS of receiving a command from the master and it takes the slave 31 µS, the transfer is likely to be worthless. This can make SPI very attractive when tight determinism is required, but unnecessarily difficult to implement otherwise. Depending on the environment, the MCU's on-board SPI peripheral might need to be used with external differential transceivers to increase signal integrity.

USB as an inter-processor communication bus

Now that more medium- to high-performance MCUs often include a USB host, it is becoming more viable as an inter-processor communication bus. Whether or not USB is viable in a given application hinges on the number of nodes, the availability of a full USB stack, and developers that can harness it. The USB virtual-comm class used in this book wasn't deterministic since it used bulk endpoints. However, interrupt-transfers can be used to achieve deterministic scheduling of transfers over USB, since they are polled by the host at a rate defined during enumeration. For example, on a high-speed USB link (which will often require an external PHY), this equates to messages up to 1 KB polled every 125 µS.

We've only scratched the surface of the possibilities for inter-processor communication in this section – there are many other options available, each with their own features, advantages, and disadvantages, depending on your project's requirements.

We now have an understanding of what multi-core and multi-processor systems are, some common topologies, and some ways of communicating between the processors. Next, let's take a step back and evaluate whether a multi-core or multi-processor design is necessary.

Choosing between multi-core and multi-processor systems

With more powerful MCUs and CPUs being announced every month, there is virtually an endless number of options to choose from. Multi-core MCUs are becoming more common. But the real question is – do you really need multiple cores or multiple processors in your design? Although they are readily available, you need to ask yourself whether using a complex MCU will ultimately help or hurt the design in the long run.

When to use multi-core MCUs

There are several cases where multi-core MCUs are an excellent fit:

- When true parallel-processing is required and PCB space is limited
- When tightly-coupled parallel threads of execution are required

A multi-core MCU may be the best option if your design is: a) space-constrained, or b) requires true parallel processing, or c) communication speed between two parallel processes is critical. Also, a multi-core MCU might be the best fit for the application if: a) it requires parallel-processing from multiple cores, and b) it can't be implemented using other hardware already present on the MCU – for example, running multiple CPU-intensive algorithms in parallel.

However, it is important to be aware of some downsides and alternatives. A multi-core MCU will likely be more challenging to replace than discrete MCUs, in both finding a replacement and porting the code. To consider alternatives, ask yourself whether the application truly needs parallel-execution at the CPU level, or is there simply a need to perform some operations in parallel (such as communication)?

If there is parallel functionality required, perhaps that can be implemented using dedicated peripheral hardware. An example is filling communication buffers using DMA connected to a hardware peripheral. That might make it possible to implement the *parallel* functionality without needing a second core.

Some potential alternatives to multi-core MCUs are as follows:

- Offloading some processing to hardware peripherals
- Ensuring DMA is used as much as possible
- Multiple MCUs

When to use multi-processor systems

Multi-processor systems are useful in a wide variety of circumstances, such as the following:

- When subsystem-reuse is possible
- When multiple teams are available to work on a large project in parallel
- When the device is large and physically dispersed
- When EMI considerations are paramount

However, while multi-processor systems are useful, they do have some potential drawbacks:

- Additional latency compared to having a single MCU.
- Real-time multi-processor communication can be complex and time-consuming to implement.
- Additional up-front planning is required to ensure proper subsystems are developed.

Summary

In this chapter, you were introduced to both multi-core and multi-processor systems and we covered some examples of each. You should now have an understanding of some of the differences between them and the things to consider when trying to decide which approach to take to your specific application. Several examples of inter-processor communication schemes were also introduced, along with some highlights and advantages of each, as they relate to embedded real-time systems.

The great thing about multi-core and multi-processor topologies is that once you have a solid understanding of the building blocks for the concurrent-system design (which we've covered), creating systems with more cores is just a matter of judiciously placing hardware where concurrent processing and abstraction will have the most impact.

Further reading

- NPX LPC54100, dual-core MCU : `https://www.nxp.com/products/LPC541XX`

16

Troubleshooting Tips and Next Steps

This chapter provides troubleshooting guidance. It's intended to help with problems that might be encountered in running the book's example-programs, and to help with problems you might encounter in developing your own FreeRTOS programs.

There are three parts to the chapter. The first part provides general debugging techniques for FreeRTOS programs. It provides a short summary of the debugging tools and techniques that are presented in the book. In addition, the FreeRTOS feature `configAssert()` is presented, along with debugging techniques to use with it.

The chapter's second part provides troubleshooting information for problems that you might encounter with using SEGGER Ozone and SystemView. These include problems from using the tools incorrectly, and from bugs in the tools themselves. The reported bugs are based on our use of the tools and their documentation (this includes our use of older versions of the tools). These problems and bugs might not be encountered in later versions of the tools, due to improvements and bug-fixes.

The chapter's third part concludes the chapter and the book. Recommendations are provided for further study and skill-development.

The chapter's sections are:

- General FreeRTOS debugging-techniques
- Troubleshooting for the debugging tools
- Next steps

General FreeRTOS debugging-techniques

This section explores some of the most useful techniques for analyzing and troubleshooting a FreeRTOS-based system. Periodically checking your system during development, as well as having a few standard steps to take when troubleshooting, can be a huge timesaver when evaluating an errant system – things don't always go as planned!

The troubleshooting techniques are presented in the following sections:

- Using assertions: `assert_param()` and `configAssert()`
- Tools for analyzing tasks and ISRs
- Monitoring memory-use
- Detecting stack-overflow

Using assertions

Assertions are excellent tools for catching conditions that simply *shouldn't happen*. They provide us with a simple means to check assumptions. There are two types of assertions that can be encountered when working with the book's example-programs: `assert_param()` and `configAssert()`. They are described in the following two sections, respectively.

assert_param()

Throughout the book's example-programs, assert_param() is used to: a) detect when the system is in an unacceptable state, and b) prevent the code from continuing to run. `assert_param()` is an STM-supplied macro that checks whether a condition is true. If the condition evaluates as false, then the function `assert_failed()` is called. That function enters a spin-loop.

`assert_param` **is described in** *Chapter 7, Running the FreeRTOS Scheduler, in the section Creating tasks and starting the scheduler*.

A special FreeRTOS flavor of the assert construct is `configAssert()`.

configAssert()

configAssert() is used throughout FreeRTOS as a way of guarding against an improperly configured system. For example, it can be used to detect when a non-interrupt version of an API is called inside an ISR. Another use is to detect when an ISR attempts to call a FreeRTOS API, but the ISR's logical priority is higher than what FreeRTOS allows.

Rather than allowing an application to run with undefined behavior, FreeRTOS will regularly test a set of assertions to ensure all prerequisites are met. On their own, these checks are helpful at preventing a system from careening completely out of control with no hope of figuring out what the problem is. Instead, the system is immediately halted when the invalid condition occurs. FreeRTOS also contains thorough documentation on the underlying reasons the assertion has failed (sometimes with links to web-based documentation).



Never *cover-up* a `configAssert()` by disabling it in any way. They are often the first notification that a serious configuration problem exists. Disabling the assertion will only compound the underlying issue, making it harder to find later.

Let's go through a conceptual example that shows what the normal symptoms are of a system that is halted by `configAssert()`, as well as the steps that can be taken to diagnose and solve the underlying issue.

Debugging a hung system with configAssert()

We'll show a series of steps for diagnosing a program that has failed due to a stack overflow. The example shown is fairly representative of what can happen when new functionality is added to a pre-existing task that was working properly. The new functionality often requires calling more functions, and each of those functions typically has local variables. As a consequence, the added function-calls will consume additional stack-space for the task.

The example-program shown here is not provided in the book's GitHub repository. The debugging steps that are shown just illustrate how such a program can be debugged. If you'd like to try these debugging techniques, some of them are applicable to `main_heap1_DeleteAttempt.c`, from *Chapter 14*. (Uncomment its call to `vTaskDelete()`. The call will fail, and FreeRTOS will call `configAssert()`.)

In this example, the problem showed up after adding calls to the SEGGER print-function within an existing task. Because there wasn't additional stack-space available, the task overflowed its stack, and the overflow corrupted the memory that was being used by another task.

Let's walk through the debugging steps that lead us to diagnose the stack overflow and identify the code that caused it.

Initially, we just know that when the system is started, it quickly becomes unresponsive, and we determine that by observing the LEDs stop blinking. We then ensure that all of the code is syntactically correct, and we confirm that when the system first starts, the LEDs are blinking as expected.

Next, it's time to connect SystemView to the running application and get some timing diagrams. The first couple of times SystemView is connected, a few events are shown, but then the system goes unresponsive:

- The LEDs stopped blinking
- No additional events were showing up in SystemView, as seen in *Figure 16.1*:

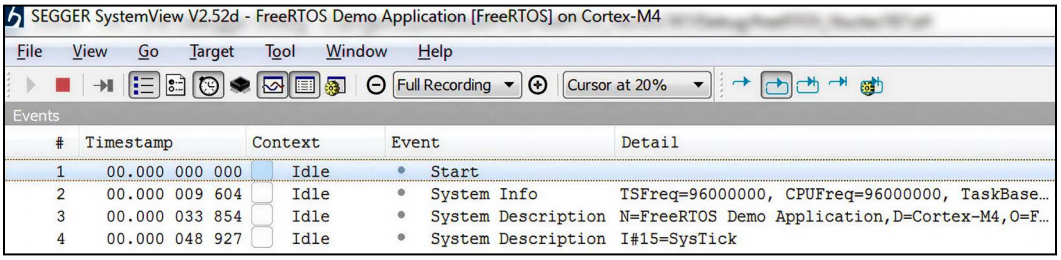


Figure 16.1 : No additional events in SystemView

Let's diagnose and solve the underlying issue in a couple of steps.

Collecting the data

Sometimes, it is tempting to take guesses as to what might be happening or make assumptions about the system. Rather than doing either of these things, we'll simply connect our debugger to the system to see what the problem is.

Since SEGGER Ozone is exceptionally good at connecting to a running system without modifying its state, we're able to connect to the hung application without disrupting anything. This allows us to start debugging an application while it is crashed, even if it is *not* being run under the debugger. This can come in very handy during product development since it allows us to run the system normally, without constantly starting it from the debugger. Let's learn how to do this:

1. Run the target code on the dev-board. If using Ozone to do that, open the Ozone project for the code, and click to **Download and reset** the firmware. Then, click on **Stop debug session**. The firmware will then start running.
2. Set up Ozone with the same code that is running on the target. This can be done by starting the Ozone project for the code. Note that the dev-board must be connected via USB.

3. After that, select **Attach to Running Program**:

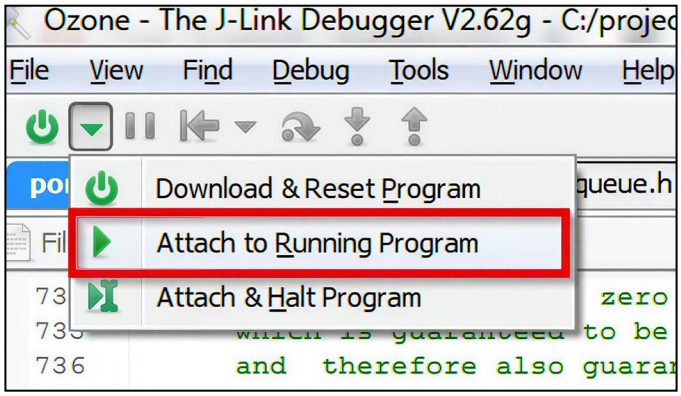


Figure 16.2 : Attaching Ozone to the running program

4. After attaching, pause execution (click on **Halt**). We're greeted with the following screen and are immediately able to make some observations:

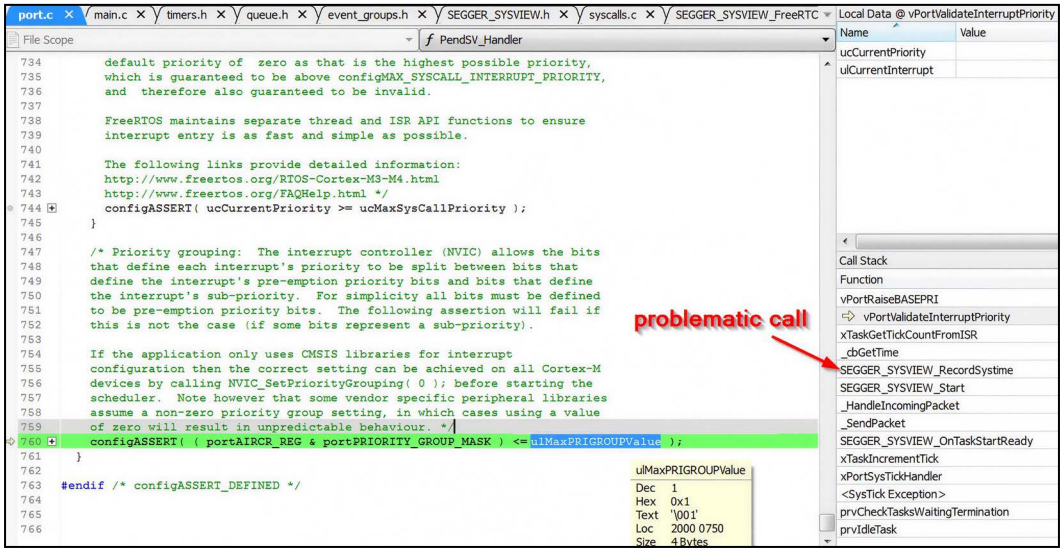


Figure 16.3 : Screen after pausing execution

We then notice the following:

- The LEDs have stopped blinking because we're spending all of our time in an infinite loop because of a failed assertion.
- By looking at the **Call Stack**, we can see that the offending function is `SEGGER_SYSVIEW_RecordSystime`, which is apparently making a call to a function called `_cbGetTime`, which in turn calls `xTaskGetTickCountFromISR`.
- Reading through the detailed comment above line 760, it sounds like there may be some misconfigured NVIC priority bits.
- The maximum acceptable value of `ulMaxPROGROUPValue` (which can be seen by hovering over the selected variable) is `1`.

Now that we know *which* assertion failed, it's time to figure out the root cause of exactly *why* it failed.

Digging deeper – Ozone data breakpoints

So far, we've determined where our processor is stuck, but we haven't uncovered anything to help us determine what needs to be changed to get the system operational again. Here are the steps we need to take to uncover the root cause of the issue:

1. Let's take a look at the assertion again. Here, our goal is to troubleshoot exactly why it is failing. Run the following instruction:

```
configASSERT( ( portAIRCR_REG & portPRIORITY_GROUP_MASK ) <=
ulMaxPRIGROUPValue );
```

2. Using Ozone's memory viewer, analyze the value of `portAIRCR_REG` in `port.c`:

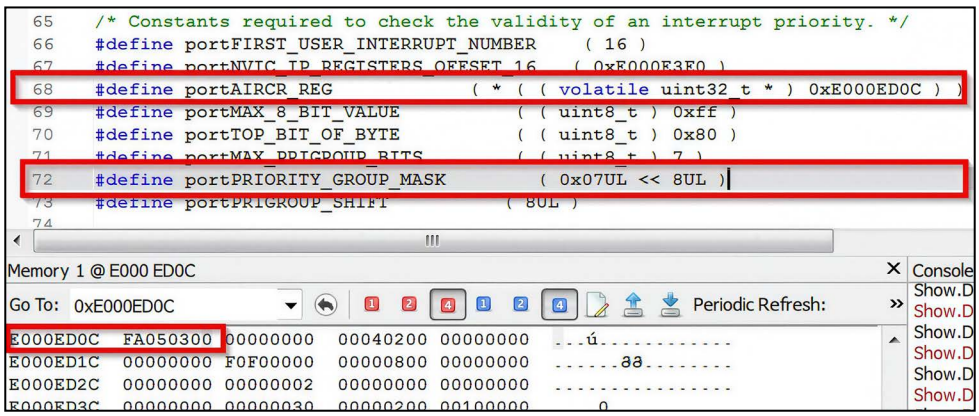


Figure 16.4 : Using Ozone's memory viewer

3. Since this is a hardcoded memory-location, we can use **Set Data Breakpoint** (right-click in the window **Break & Tracepoints**). It will pause execution each time the memory location is written. This can be a quick way to track down all of the ways a variable is accessed, without attempting to search through the code:

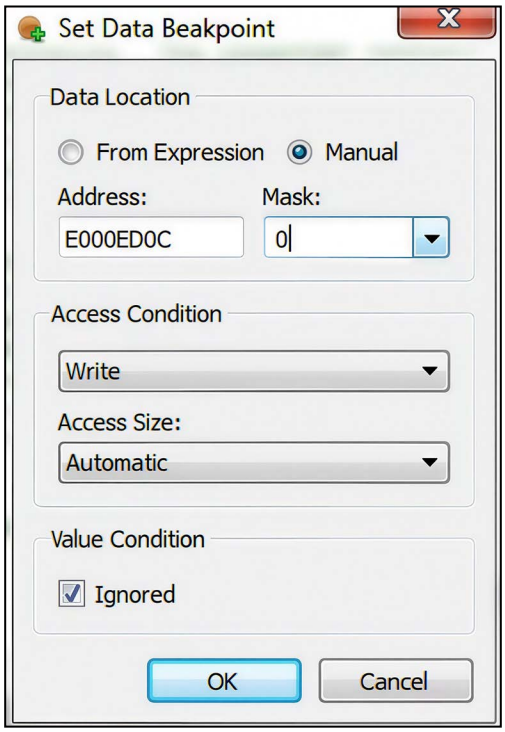


Figure 16.5 : Setting a data breakpoint

4. Upon restarting the MCU, the write breakpoint is immediately hit. Although the program-counter is pointing to `HAL_InitTick`, the actual data write to the `0xE000ED0C` address was done in the previous function, that is, `HAL_NVIC_SetPriorityGrouping`. This is exactly what we expect since the assert is related to interrupt priority groups:

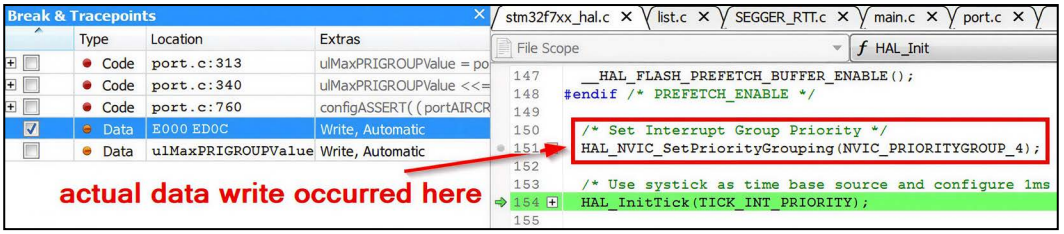


Figure 16.6 : The write breakpoint is hit

1. Some quick searching through the code for `NVIC_PRIORITYGROUP_4` reveals the following comment in `stm32f7xx_hal_cortex.c`:

```
* @arg NVIC_PRIORITYGROUP_4: 4 bits for preemption priority
*                            0 bits for subpriority
```



From …`FreeRTOS\Source\portable\GCC\ARM_CM7\r0p1\port.c`:

/* Priority grouping: The interrupt controller (NVIC) allows the bits that define each interrupt's priority to be split between bits that define the interrupt's preemption (group) priority-bits, [as well as the] bits that define the interrupt's sub-priority. For simplicity, all bits must be defined to be preemption (group) priority bits. The following assertion will fail if this is not the case (if some bits represent a sub-priority).

...*/

Based on this information, there should be 0 bits for `subpriority`. So, why was the value of the priority bits in `portAIRCR_REG` non-zero?

From the ARM® Cortex® -M7 Devices Generic User Guide, we can see that to achieve 0 bits of subpriority, the value of the **AIRCR** register masked with `0x00000700` must read as 0 (it had a value of 3 when we looked at the value in memory):

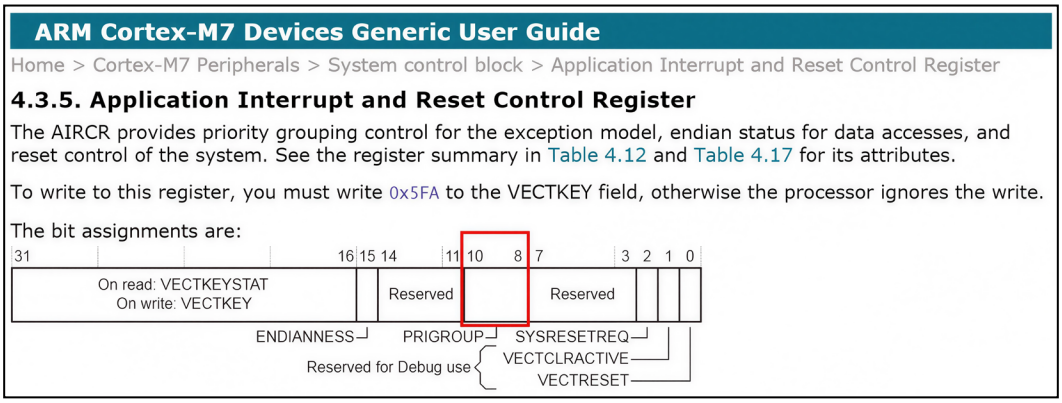


Figure 16.7 : AIRCR register specifications

Here is the explanation for `PRIGROUP` in the same manual. Notice that `PRIGROUP` must be set to `0b000` for 0 subpriority bits:

Binary point

The PRIGROUP field indicates the position of the binary point that splits the PRI_*n* fields in the Interrupt Priority Registers into separate *group priority* and *subpriority* fields. Table 4.18 shows how the PRIGROUP value controls this split. Implementations having PRI_*n* fields of less than 8 bits treat the least-significant bits as zero.

Table 4.18. Priority grouping

	Interrupt priority level value, PRI_*N*[7:0]			Number of	
PRIGROUP	**Binary point[a]**	**Group priority bits**	**Subpriority bits**	**Group priorities[b]**	**Subpriorities[b]**
0b000[c]	bxxxxxxx.y	[7:1]	[0]	128	2
0b001[c]	bxxxxxx.yy	[7:2]	[1:0]	64	4
0b010[c]	bxxxxx.yyy	[7:3]	[2:0]	32	8
0b011[c]	bxxxx.yyyy	[7:4]	[3:0]	16	16
0b100	bxxx.yyyyy	[7:5]	[4:0]	8	32
0b101	bxx.yyyyyy	[7:6]	[5:0]	4	64
0b110	bx.yyyyyyy	[7]	[6:0]	2	128
0b111	b.yyyyyyyy	None	[7:0]	1	256

[a] PRI_*n*[7:0] field showing the binary point. x denotes a group priority field bit, and y denotes a subpriority field bit.

Figure 16.8 : PRIGROUP field specifications

This certainly warrants further investigation... why was the value of `PRIOGROUP` 3 instead of 0? Let's take another look at that `configAssert()` line:

```
configASSERT( ( portAIRCR_REG & portPRIORITY_GROUP_MASK ) <=
ulMaxPRIGROUPValue );
```

Note the following definition of `ulMaxPRIOGROUPValue` in `port.c`. It is defined as *static*, which means it has a permanent home in memory:

```
#if( configASSERT_DEFINED == 1 )
static uint8_t ucMaxSysCallPriority = 0;
static uint32_t ulMaxPRIGROUPValue = 0;
```

Let's set up another data breakpoint for `ulMaxPRIGROUPValue` and restart the MCU again, but this time, we'll watch each time it is accessed:

- As expected, something was accessed by the `BaseType_t xPortStartScheduler(void)` function in `port.c`

- The curious part about the data-access breakpoint is that it is hit when the program counter is inside `SEGGER_RTT.c`, which doesn't look right since `ulMaxPRIGROUPValue` is privately scoped to `xPortStartScheduler` in `port.c`
- Looking at the debugger – the problem is staring right at us:
    - The `ulMaxPRIGROUPValue` static variable is being stored in `0x2000 0750`.
    - The data write breakpoint was hit with the stack pointer at `0x200 0740`.
    - The stack has been overrun:

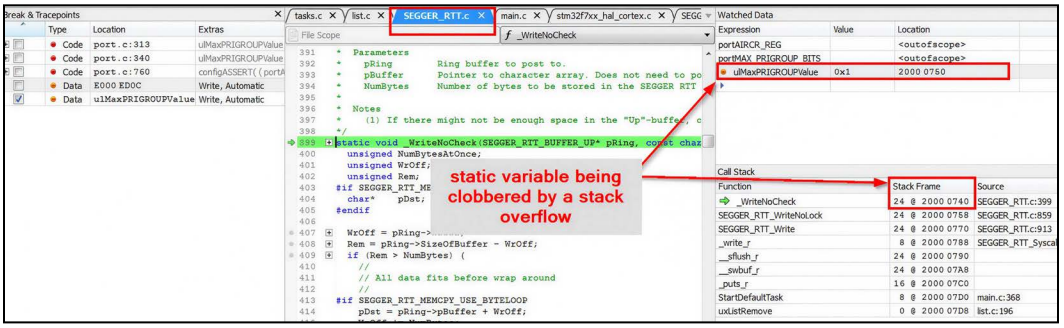


Figure 16.9 : Stack overflow

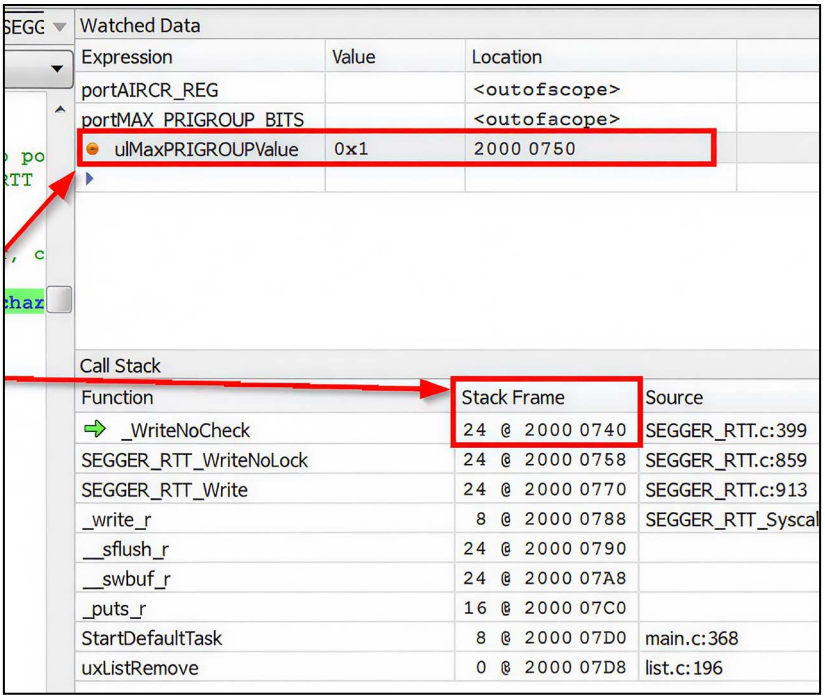


Figure 16.9a: The right side of figure 16.9

We've just uncovered a stack overflow. It manifested itself as a write into a static variable, which happened to trigger a `configAssert` in an unrelated part of the system. This type of wildly unexpected behavior is a common side-effect of stack overflows.

Currently, the minimum values of each task's stack-size in `main.c` has been set to 128 words (1 word = 4 bytes), so increasing this to 256 words (1 KB) gives us plenty of headroom.

The problem in this example would have likely been caught if we had the stack-overflow hooks set up – it would have certainly been caught if the MPU port was being used.

The next three sections summarize the debugging tools presented in the book.

Tools for analyzing tasks and ISRs

Being able to get a clear understanding of what all the threads in a system are doing is a huge help – for novices and experts alike. Tooling is especially helpful for this. Using a visualization tool such as SEGGER SystemView or Percepio Tracealyzer can be invaluable in understanding interactions between various tasks and interrupts in a system.

Having a FreeRTOS-aware debugger is also a huge help since it allows us to stack the analysis of multiple tasks. This debugger can be part of your IDE, or a standalone debugger such as SEGGER Ozone.

In the present book, the following sections explain how to use Ozone and SystemView:

- *Chapter 3 Introducing the Development Tools*, section *The debugging tools: J-Link, Ozone and SystemView*
- *Chapter 6 Understanding RTOS Tasks*, section *Using SystemView with FreeRTOS*
- *Chapter 7 Running the FreeRTOS Scheduler*, sections:
    - *Additional Ozone features*
    - *Using SystemView to analyze task execution*
    - *Troubleshooting task startup problems*

Monitoring memory-use

Memory use is a very important aspect to consider when using an RTOS. Unlike a super-loop with a single stack – which, along with the heap, would consume whatever RAM was *left over* – each FreeRTOS task's stack needs to be explicitly sized.

In *Chapter 14, FreeRTOS Memory Management*, section *Keeping an eye on stack space*, we showed you how to observe the available stack space, as well as how to implement hooks if an overflow was detected.

If your application is using dynamic memory-allocation, you should strongly consider enabling and implementing the failed `malloc` hooks provided by FreeRTOS. This was covered in *Chapter 14*, in the section *Keeping an eye on heap space*.

Reference documents on memory-use are listed in *Appendix B, Reference Information*, under the section for *Chapter 14*.

STM32CubeIDE has tools for memory analysis. They are documented in the IDE's user guide, in these sections:

- 2.2.1.6 "Linker script", Table 3 "Memory map layout"
- 2.4 "Build Analyzer" (shows memory layout for the elf)
- 2.6 "Static Stack Analyzer" (shows stacks for the elf)

Detecting stack-overflow

If you have a memory protection unit available, it is an excellent idea to make use of it. It can detect access violations, such as stack overflows, with better reliability than any of the software-based solutions. (See *Chapter 14*, in the section *Using a memory protection unit.*)

Another way of keeping an eye on the stack is to set-up stack monitoring, which was also covered in *Chapter 14*, in the section *Keeping an eye on stack space*.

Troubleshooting for the debugging tools

Troubleshooting Ozone problems

There are two sections on Ozone troubleshooting, below. The first section describes problems that can be encountered when using Ozone. The second section lists Ozone's console output for two of the book's example-programs. The console output is from the programs' successful operation. If Ozone does not start successfully, those listings can be used as a reference-point for troubleshooting.

Possible problems

Let's begin by looking at two possible problems with Ozone, and how we can solve them.

Problem: Ozone hangs when starting a debug-session

- **Problem**:

  In Ozone, after **Download and Reset Program** is run, a debug session is not started and Ozone hangs. Ozone's icon **Resume Program Execution** does not appear. The last console message is "`Debug.Start();`".

- **Solution**:

  The problem may be due to SystemView not shutting down properly the last time it was run. Disconnect the USB cable for the board, to restart it. Windows Task Manager can be used to see if Ozone and SystemView tasks are running, and to end them. Two examples are the tasks `JLinkGUIServer.exe` and `SystemView`. An alternative solution is to reboot the computer that's running Ozone.

Problem: Ozone will not close

- **Problem**:

  The Ozone application will not close, even after clicking on Ozone's **Exit** and on the window-close icon (**X**).

- **Solution**:

  Disconnect the board's USB cable, and use Windows Task Manager to end the Ozone task.

Console output for successful operation

In this section, Ozone console-output is shown for an example-program that does not use FreeRTOS, and for an example-program that does. The console output is for Ozone starting successfully.

If Ozone does not start successfully, these listings can be used as a reference point for troubleshooting. Deviations from the listing can indicate what went wrong, and where.

Console output from Chapter_03 (FreeRTOS is not used)

The listing below is the Ozone console-output for the `Chapter_03` example-program. That program does not use FreeRTOS. (For programs that use FreeRTOS, the console output is different.) The console-output was obtained from opening the program's Ozone project file, then clicking on **Download and Reset Program**, and **Resume Program Execution**.

```
Disabled output of control characters
SEGGER Ozone - The J-Link Debugger V3.38f
J-Link software found at: C:/Program Files/SEGGER/Ozone V3.38f/JLink_x64.
dll
File.Open ("C:/projects/packtBookRTOS2E/Chapter_03/Ozone-project-file.
jdebug");
Project.SetTraceSource ("SWO");
Project.SetDevice ("STM32F767ZI");
Target core support plugin loaded.: C:/Program Files/SEGGER/Ozone V3.38f/
Plugins/Core/CorePluginARM.dll
```

```
Project.SetHostIF ("USB", "");
Project.SetTargetIF ("SWD");
Project.SetTIFSpeed ("4 MHz");
Project.AddSvdFile ("C:\ST\STM32CubeIDE_1.18.1\STM32CubeIDE\plugins\com.
st.stm32cube.ide.mcu.productdb.debug_2.2.100.202502251557\resources\cmsis\
STMicroelectronics_CMSIS_SVD\STM32F767.svd");
Project.AddSvdFile ("C:\Program Files\SEGGER\Ozone V3.38f\Config\CPU\
Cortex-M7F.svd");
Project.SetRTT (0);
File.Open ("C:/projects/packtBookRTOS2E/Chapter_03/Debug/Chapter_03.elf");
File.Open: completed in 655 ms
Program segments:
  Address         Size         Code      RO Data      RW Data      ZI Data
Flg
---------    ---------    ---------    ---------    ---------    --------
-    ---
 08000000       19 072       18 384          688            0            0
RWE
 20000000        3 720            0            0           12        3 708
RW
 20000E88        1 536            0            0        1 536            0
RW
---------    ---------    ---------    ---------    ---------    --------
-    ---
   Total:       24 328       18 384          688        1 548        3 708
---------    ---------    ---------    ---------    ---------    --------
-    ---
For further information on ELF file data sections, execute command Elf.
PrintSectionInfo(0).
Debug.ReadIntoInstCache: updated instruction information within 1 code
ranges (0x080001F8-0x080049C8)
Debug.SetConnectMode (CM_DOWNLOAD_RESET);
Debug.Start();
Device "STM32F767ZI" selected.
InitTarget() start
SWD selected. Executing JTAG -> SWD switching sequence.
DAP initialized successfully.
InitTarget() end - Took 28.6ms
```

```
Found SW-DP with ID 0x9BA02477
DPv0 detected
CoreSight SoC-400 or earlier
Scanning AP map to find all available APs
AP[1]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x74770001, ADDR: 0x00000000)
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FD000
CPUID register: 0x411FC270. Implementer code: 0x41 (ARM)
Cache: L1 I/D-cache present
Found Cortex-M7 r1p0, Little endian.
FPUnit: 8 code (BP) slots and 0 literal slots
CoreSight components:
ROMTbl[0] @ E00FD000
[0][0]: E00FE000 CID B105100D PID 000BB4C8 ROM Table
ROMTbl[1] @ E00FE000
[1][0]: E00FF000 CID B105100D PID 000BB4C7 ROM Table
ROMTbl[2] @ E00FF000
[2][0]: E000E000 CID B105E00D PID 000BB00C SCS-M7
[2][1]: E0001000 CID B105E00D PID 000BB002 DWT
[2][2]: E0002000 CID B105E00D PID 000BB00E FPB-M7
[2][3]: E0000000 CID B105E00D PID 000BB001 ITM
[1][1]: E0041000 CID B105900D PID 001BB975 ETM-M7
[0][1]: E0040000 CID B105900D PID 000BB9A9 TPIU-M7
I-Cache L1: 16 KB, 256 Sets, 32 Bytes/Line, 2-Way
D-Cache L1: 16 KB, 128 Sets, 32 Bytes/Line, 4-Way
Connected to target device.
J-Link/J-Trace serial number: 779371901
Reset type: NORMAL (https://wiki.segger.com/J-Link_Reset_Strategies)
Reset: Halt core after reset via DEMCR.VC_CORERESET.
Reset: Reset device via AIRCR.SYSRESETREQ.
Elf.GetBaseAddr(); // returns 0x8000000
Target.ReadU32 (0x08000000); // returns 0x4, data is 0x20080000
Target.SetReg ("SP", 0x20080000);
Target.ReadU32 (0x08000004); // returns 0x4, data is 0x8004481
Target.SetReg ("PC", 0x8004481);
```

```
J-Link: Flash download: Bank 0 @ 0x08000000: 1 range affected (32768
bytes)
J-Link: Flash download: Total: 0.884s (Prepare: 0.089s, Compare: 0.175s,
Erase: 0.363s, Program & Verify: 0.226s, Restore: 0.029s)
J-Link: Flash download: Program & Verify speed: 141 KB/s
Elf.GetBaseAddr(); // returns 0x8000000
Target.ReadU32 (0x08000000); // returns 0x4, data is 0x20080000
Target.SetReg ("SP", 0x20080000);
Target.ReadU32 (0x08000004); // returns 0x4, data is 0x8000C21
Target.SetReg ("PC", 0x8000C21);
Memory map 'after startup completion point' is active
Startup complete (PC=0x080005F8)
SWO_EnableTarget() start
SWO_EnableTarget() end - Took 1.26ms
Debug.Continue();
```

Console output from Chapter_06 (FreeRTOS is used)

The listing below is the Ozone console-output for the `Chapter_06` example-program. That program uses FreeRTOS. The console-output was obtained by opening the program's Ozone project file, then clicking on **Download and Reset Program**, and **Resume Program Execution**.

```
Disabled output of control characters
SEGGER Ozone - The J-Link Debugger V3.38f
J-Link software found at: C:/Program Files/SEGGER/Ozone V3.38f/JLink_x64.
dll
File.Open ("C:/projects/packtBookRTOS2E/Chapter_06/Ozone-project-file.
jdebug");
Project.SetTraceSource ("SWO");
Project.SetDevice ("STM32F767ZI");
Target core support plugin loaded.: C:/Program Files/SEGGER/Ozone V3.38f/
Plugins/Core/CorePluginARM.dll
Project.SetHostIF ("USB", "");
Project.SetTargetIF ("SWD");
Project.SetTIFSpeed ("4 MHz");
Project.SetOSPlugin ("FreeRTOSPlugin_CM7");
File name resolved: "FreeRTOSPlugin_CM7.js" was found at "C:/Program
Files/SEGGER/Ozone V3.38f/Plugins/OS/FreeRTOSPlugin_CM7.js"
```

```
RTOS awareness plugin loaded: C:/Program Files/SEGGER/Ozone V3.38f/
Plugins/OS/FreeRTOSPlugin_CM7.js.
Project.AddSvdFile ("C:\ST\STM32CubeIDE_1.18.1\STM32CubeIDE\plugins\com.
st.stm32cube.ide.mcu.productdb.debug_2.2.100.202502251557\resources\cmsis\
STMicroelectronics_CMSIS_SVD\STM32F767.svd");
Project.AddSvdFile ("C:\Program Files\SEGGER\Ozone V3.38f\Config\CPU\
Cortex-M7F.svd");
Project.SetRTT (0);
File.Open ("C:/projects/packtBookRTOS2E/Chapter_06/Debug/Chapter_06.elf");
File.Open: completed in 654 ms
Program segments:
  Address         Size         Code      RO Data      RW Data      ZI Data
Flg
---------    ---------    ---------    ---------    ---------    --------
-    ---
 08000000       32 976       31 680        1 296            0            0
RWE
 20000000       21 512            0            0           40       21 472
RW
 20005408        1 536            0            0        1 536            0
RW
---------    ---------    ---------    ---------    ---------    --------
-    ---
   Total:       56 024       31 680        1 296        1 576       21 472
---------    ---------    ---------    ---------    ---------    --------
-    ---
For further information on ELF file data sections, execute command Elf.
PrintSectionInfo(0).
Debug.ReadIntoInstCache: updated instruction information within 1 code
ranges (0x08000200-0x08007DC0)
Debug.SetConnectMode (CM_DOWNLOAD_RESET);
Debug.Start();
Device "STM32F767ZI" selected.
InitTarget() start
SWD selected. Executing JTAG -> SWD switching sequence.
DAP initialized successfully.
InitTarget() end - Took 19.0ms
Found SW-DP with ID 0x9BA02477
DPv0 detected
```

```
CoreSight SoC-400 or earlier
Scanning AP map to find all available APs
AP[1]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x74770001, ADDR: 0x00000000)
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FD000
CPUID register: 0x411FC270. Implementer code: 0x41 (ARM)
Cache: L1 I/D-cache present
Found Cortex-M7 r1p0, Little endian.
FPUnit: 8 code (BP) slots and 0 literal slots
CoreSight components:
ROMTbl[0] @ E00FD000
[0][0]: E00FE000 CID B105100D PID 000BB4C8 ROM Table
ROMTbl[1] @ E00FE000
[1][0]: E00FF000 CID B105100D PID 000BB4C7 ROM Table
ROMTbl[2] @ E00FF000
[2][0]: E000E000 CID B105E00D PID 000BB00C SCS-M7
[2][1]: E0001000 CID B105E00D PID 000BB002 DWT
[2][2]: E0002000 CID B105E00D PID 000BB00E FPB-M7
[2][3]: E0000000 CID B105E00D PID 000BB001 ITM
[1][1]: E0041000 CID B105900D PID 001BB975 ETM-M7
[0][1]: E0040000 CID B105900D PID 000BB9A9 TPIU-M7
I-Cache L1: 16 KB, 256 Sets, 32 Bytes/Line, 2-Way
D-Cache L1: 16 KB, 128 Sets, 32 Bytes/Line, 4-Way
Connected to target device.
J-Link/J-Trace serial number: 779371901
Reset type: NORMAL (https://wiki.segger.com/J-Link_Reset_Strategies)
Reset: Halt core after reset via DEMCR.VC_CORERESET.
Reset: Reset device via AIRCR.SYSRESETREQ.
Elf.GetBaseAddr(); // returns 0x8000000
Target.ReadU32 (0x08000000); // returns 0x4, data is 0x20080000
Target.SetReg ("SP", 0x20080000);
Target.ReadU32 (0x08000004); // returns 0x4, data is 0x8000C21
Target.SetReg ("PC", 0x8000C21);
J-Link: Flash download: Bank 0 @ 0x08000000: 1 range affected (65536
bytes)
```

```
J-Link: Flash download: Total: 1.343s (Prepare: 0.068s, Compare: 0.148s,
Erase: 0.712s, Program & Verify: 0.385s, Restore: 0.028s)
J-Link: Flash download: Program & Verify speed: 166 KB/s
Elf.GetBaseAddr(); // returns 0x8000000
Target.ReadU32 (0x08000000); // returns 0x4, data is 0x20080000
Target.SetReg ("SP", 0x20080000);
Target.ReadU32 (0x08000004); // returns 0x4, data is 0x8007CC1
Target.SetReg ("PC", 0x8007CC1);
Memory map 'after startup completion point' is active
Startup complete (PC=0x080079C4)
SWO_EnableTarget() start
SWO_EnableTarget() end - Took 1.27ms
Debug.Continue();
```

Troubleshooting SystemView problems

There are two sections on SystemView troubleshooting, below. The first section describes problems that can be encountered when using SystemView. The second section lists SystemView's console output for one of the book's example programs. The console output is from the program's successful operation. If SystemView does not start successfully, that listing can be used as a reference-point for troubleshooting.

Possible problems

This section describes four possible problems with SystemView, and how to solve them. The problems reported are based on our experience with SystemView, and they are subject to change, e.g., due to SystemView enhancements and bug fixes. Let's dive in!

Problem: SystemView Recorder overflows, or does not record all events

- **Problem**:

  When SystemView Recorder is run, some events might not be displayed in SystemView. For example, a pop-up message may be displayed, stating, **SystemView : overflow events recorded**. Also, in the SystemView Timeline, some areas may be blocked-out in solid red, to show when data was lost due to overflow.



Figure 16.10 : Dropped data in SystemView

- **Solution**:

  The problem is that the firmware is trying to send more data to the development-computer than is possible. The SystemView user-manual lists causes of overflow and possible solutions. They include:

  - J-Link is being kept busy by a debugger, so J-Link cannot read the data fast enough from the MCU. This can happen with Ozone. One solution is to end the Ozone debug-session before running the SystemView app in Record Mode.
  - The application generates too many events to fit into the SystemView RTT-buffer on the MCU. The RTT-buffer holds the events recorded by the firmware, which are to be sent to the SystemView application. This includes FreeRTOS system-events recorded by SystemView, e.g., task context-switches. One way to reduce the number of events is to reduce the number of calls to `SEGGER_SYSVIEW_PrintfHost()`.
  - The "target interface" speed is too low to read the data fast enough. This apparently refers to the interface between the MCU and the J-Link debug-probe. One solution can be to upgrade to a J-Link debug-probe with a faster interface.

For the RTT-buffer on the MCU, the buffer's size is configurable. If the solutions listed above are not sufficient, it might be worth trying a larger RTT buffer. It is important to note that increasing the buffer's size will decrease the memory available to other system components.

Problem: SEGGER_SYSVIEW_PrintfHost() does not work as expected

`SEGGER_SYSVIEW_PrintfHost()` is a SystemView API that is called in the firmware. For brevity, the function will be referred to as `S...PrintfHost()`. It generates a message that is displayed by the SystemView application, e.g., in the Terminal window. The interface for `S...PrintfHost()`is modeled after `printf()`. Documentation for `S...PrintfHost()` is in the SystemView users guide.

- **Problem**:

  In our experience, the SystemView users-guide omits some information that is needed to use `S...PrintfHost()`. Although `S...PrintfHost()`is modeled after `printf()`, there are significant differences between them, and the differences are not adequately specified.

  `printf()` uses *format-placeholders* to specify string-formatting, e.g., `%d`. `S...PrintfHost()` only supports a subset of the `printf()` format-placeholder types, e.g., `%d` is supported, but not `%s`. The SystemView users-guide does not fully specify which types are supported and which are not.

  Also, when `S...PrintfHost()` is used incorrectly, there may be no syntax warning-messages, and the SystemView application may display incorrect data.

- **Solution**:

  Most of those problems can be avoided by only using the format-placeholder-types that are supported by `S...PrintfHost()`. From our investigation, it appears the following types are supported: `%c` (character), `%d` (signed decimal integer), `%p` (pointer address), `%u` (unsigned decimal integer), and `%x` (unsigned hexadecimal integer). It appears that the following types are not supported: `%f` (floating point), `%F` (floating point), `%s` (string), and `%%` (generates "%"). This list of supported and unsupported types is not intended to be complete.

  If you're considering using `sprintf()` to augment `S...PrintfHost()`, beware that `sprintf()` can be problematic with FreeRTOS, e.g., it may not be thread-safe.

- **Problem**:

  In our experience, there is a problem with `S...PrintfHost()` when the output-string has a binary-zero before the end of the string. In that case, the binary-zero is interpreted as the end of the string, when displayed by SystemView. Any characters after the first binary-zero will not be displayed. This behavior is different than `printf()` when output is sent to the console, e.g., on Ubuntu. There, binary-zeros before the end of the string are not interpreted as the end of the string. The SystemView users-guide does not describe this discrepancy between `S...PrintfHost()` and `printf()`. The discrepancy is illustrated in the following example.

  With `printf("12%c45", '3')`, the string displayed is "12345". And, with `printf("12%c45", (char)0)`, the string displayed is "1245".

For the latter example, if those arguments are used with `S...PrintfHost()`, the string displayed in SystemView is different. For `S...PrintfHost("12%c45", (char)0)`, the string displayed in SystemView is "12". The characters "45" are not displayed, as they would be with `printf()`.

- **Solution**:

  When using `S...PrintfHost()`, the output-string's first binary zero will be treated as the end of the string, when displayed by SystemView. A work-around is to substitute another value for the binary-zeros before the string's null-terminator, e.g., substitute the space character (" ").

Problem: SystemView cannot be opened or closed

- **Problem**:

  There are situations in which SystemView cannot be opened or closed. When this problem has occurred, it appeared to be related to closing SystemView after encountering problems with it, such as hanging or failing to connect.

- **Solution**:

  Disconnect the USB cable for the board, to restart it. Open Windows Task Manager and end SystemView tasks, including these, if present: `SystemView`, `JLinkGUIServer.exe`.

Problem: SystemView hangs

- **Problem**:

  We encountered two instances in which SystemView hung. One instance was just after clicking-on "**Start Recording**". The other instance was after SystemView had been recording for an extended period.

- **Solution**:

  These problems can be caused by incorrect SystemView settings. The first instance described was apparently due to the Ozone project-file (`*.jdebug`) not having the line: "`Project.SetRTT(0);`". The other instance may be due to specifying an incorrect **Interface Speed** in SystemView's **Recorder Configuration**. In general, if SystemView hangs, it is prudent to check that Ozone and SystemView are configured correctly.

Console output for successful operation

The listing below is the SystemView console-output for the `Chapter_03` example-program. That program does not use FreeRTOS. (For programs that use FreeRTOS, the console-output is the same.) The console-output is displayed in the SystemView **Log** window.

If SystemView does not start successfully, this listing can be used as a reference point for troubleshooting. Deviations from this listing can indicate what went wrong, and where.

```
20:20:53 - Command listener started on port 19050.
20:20:53 - SEGGER SystemView V3.60d started @ 12. Jun 2025 20:20:53
20:21:53 - JLink: Device "STM32F767ZI" selected.
20:21:53 - JLink: InitTarget() start
20:21:53 - JLink: SWD selected. Executing JTAG -> SWD switching sequence.
20:21:53 - JLink: DAP initialized successfully.
20:21:53 - JLink: InitTarget() end - Took 40.4ms
20:21:53 - JLink: Found SW-DP with ID 0x9BA02477
20:21:53 - JLink: DPv0 detected
20:21:53 - JLink: CoreSight SoC-400 or earlier
20:21:53 - JLink: Scanning AP map to find all available APs
20:21:53 - JLink: AP[1]: Stopped AP scan as end of AP map has been reached
20:21:53 - JLink: AP[0]: AHB-AP (IDR: 0x74770001, ADDR: 0x00000000)
20:21:53 - JLink: Iterating through AP map to find AHB-AP to use
20:21:53 - JLink: AP[0]: Core found
20:21:53 - JLink: AP[0]: AHB-AP ROM base: 0xE00FD000
20:21:53 - JLink: CPUID register: 0x411FC270. Implementer code: 0x41 (ARM)
20:21:53 - JLink: Cache: L1 I/D-cache present
20:21:53 - JLink: Found Cortex-M7 r1p0, Little endian.
20:21:53 - JLink: FPUnit: 8 code (BP) slots and 0 literal slots
20:21:53 - JLink: CoreSight components:
20:21:53 - JLink: ROMTbl[0] @ E00FD000
20:21:53 - JLink: [0][0]: E00FE000 CID B105100D PID 000BB4C8 ROM Table
20:21:53 - JLink: ROMTbl[1] @ E00FE000
20:21:53 - JLink: [1][0]: E00FF000 CID B105100D PID 000BB4C7 ROM Table
20:21:53 - JLink: ROMTbl[2] @ E00FF000
20:21:53 - JLink: [2][0]: E000E000 CID B105E00D PID 000BB00C SCS-M7
20:21:53 - JLink: [2][1]: E0001000 CID B105E00D PID 000BB002 DWT
20:21:53 - JLink: [2][2]: E0002000 CID B105E00D PID 000BB00E FPB-M7
20:21:53 - JLink: [2][3]: E0000000 CID B105E00D PID 000BB001 ITM
```

```
20:21:53 - JLink: [1][1]: E0041000 CID B105900D PID 001BB975 ETM-M7
20:21:53 - JLink: [0][1]: E0040000 CID B105900D PID 000BB9A9 TPIU-M7
20:21:53 - JLink: I-Cache L1: 16 KB, 256 Sets, 32 Bytes/Line, 2-Way
20:21:53 - JLink: D-Cache L1: 16 KB, 128 Sets, 32 Bytes/Line, 4-Way
20:22:01 - TRACE START Event recorded.
```

Next steps

In this final chapter, we covered a few tips that will help smooth out some of the bumps in the road on your RTOS journey.

Well, that's it, folks! We hope you've enjoyed this hands-on introduction to developing real-time embedded systems using FreeRTOS.

Assuming you've already run the book's example-programs, a good next-step in mastering FreeRTOS is to work through Richard Barry's book, *Mastering the FreeRTOS Real-Time Kernel*. (It is freely available at `FreeRTOS.org`). It presents further details about FreeRTOS, with examples for each of the APIs.

To experiment with FreeRTOS on the book's dev-board, you'll need an IDE-project with FreeRTOS installed on it. How to create such an IDE-project is described in *Appendix C, Creating FreeRTOS Projects, and Installing FreeRTOS*. Among the book's example-programs, there is an IDE-project `FreeRTOS_project`, and FreeRTOS is installed on it. *Appendix C* describes how `FreeRTOS_project` can be copied for use with your FreeRTOS experiments. *Appendix C* also describes how to create a new IDE-project and install FreeRTOS on it.

Our book provides skills in working with FreeRTOS, an ST IDE, SEGGER debugging tools, and an STM32 MCU. These skills will enable you to continue with FreeRTOS, in working on experiments and in developing operational systems.

We'd love to hear about how you've applied what you've learned in this book – give us a shout on our Discord channel (`https://packt.link/embedded`)! If you've enjoyed this book or have ideas for additional features you'd like to see, please consider leaving a review.

Appendices

Reference Information and FreeRTOS Installation

The appendices provide reference information for the book's development tools and dev-board. Also, information is provided on how to install FreeRTOS. *Appendix A* provides a quick reference ("cheat sheet") for the development tools that are used with the book's example-programs. *Appendix B* lists the many reference documents for the book's example-programs and hardware, for example, user guides. *Appendix C* describes how to create an IDE-project with FreeRTOS installed. To experiment with FreeRTOS on the book's dev-board, you'll need an IDE-project with FreeRTOS. An IDE-project is provided among the book's example-programs , and instructions for copying that IDE-project are provided. *Appendix C* also contains information on how to create a new IDE-project and install FreeRTOS.

The following are the appendices that are provided:

- *Appendix A, Tools Quick Reference*
- *Appendix B, Reference Information*
- *Appendix C, Creating FreeRTOS Projects and Installing FreeRTOS*

Appendix A: Tools Quick-Reference

This appendix provides a quick-reference ("cheat-sheet") for the development-tools used in the book. Most of this information is excerpts from chapter 3.

STM32CubeIDE

The IDE workspace and projects

- A **workspace** consists of a set of projects.
- A **project** is a directory that contains one or more programs.
- For each chapter that has example programs, the chapter's programs are put in an IDE project. We'll refer to these IDE-projects as **chapter-projects**.
- For the chapter-projects, their project-directories are stored in the book's code-repository: `https://github.com/PacktPublishing/Hands-On-RTOS-with-Microcontrollers-Second-Edition`
- The **example-programs' workspace** holds the chapter-projects, e.g., at `C:\projects\packtBookRTOS2E`.
- The IDE's prompt to select a workspace is at: **File | "Switch Workspace" | Other...**
- In the example-programs' workspace, the chapter-projects are imported from the code-repository.
- The IDE's **Project Explorer** shows the chapter-projects.

Source-code files

For a chapter-project, the user-written code is here:

- With a single build-configuration, it's typically in the directory `Core\Src`, e.g., `Chapter_3\Core\Src\main.c`.
- With multiple build-configurations, it's typically in the directory `Src`, e.g., `Chapter_07\Src\main_taskCreation.c`.

Building a project

- Activate the build-configuration to be used:
    - In the Project Explorer, right-click on the IDE-project, and select "**Build Configurations**" | "**Set Active**".
- Build the active build-configuration:
    - In the Project Explorer, right-click on the IDE-project, and select **Build Project**.
- The created `*.elf` file is stored here:
    - With a single build-configuration, it's typically in the project's `Debug` directory, e.g., `Chapter_3\Debug\Chapter_3.elf`.
    - With multiple build-configurations, it's typically in a directory with the build-configuration's name, e.g., `Chapter_07\TaskCreationBuild\Chapter_07_TaskCreationBuild.elf`.

Ozone

How to debug a program while it is running on the dev-board:

1. Connect the USB cable to the board.
2. Copy an Ozone project-file to the IDE project-directory, e.g., to the directory `Chapter_03`.
    - The book's template for project-files can be copied. It is in the example-programs' directory `templates`: `templates\Ozone-project-file.jdebug`.
    - Alternatively, a previously-configured project-file can be copied, e.g., `Chapter_03\Ozone-project-file.jdebug`.
3. Configure the Ozone project-file for the program to be debugged:
    - The template's top-most block-comment describes how to configure the file.
    - Edit the template-copy to specify the three needed file-paths, as described in the template's code-comments.
    - If the IDE project has multiple build-configurations, be sure to specify the correct `*.elf` file.
    - In the template-copy, uncomment the line needed for FreeRTOS, if FreeRTOS is being used.
4. Open Ozone, and load the `*.jdebug` file:
    - Click on **Open Existing Project** and select the `*.jdebug` file.

5. Load the `*.elf` file onto the board:
    - On the toolbar, click on the drop-down button, and select **Download & Reset Program**.
6. Run the program (`*.elf` file):
    - On the toolbar, click on the play button (**Resume program execution**).

SystemView

How to run the SystemView app:

1. Start the SystemView app.
2. If the "Recorder Configuration" has not been set-up:
    - Open the **Recorder Configuration** window by clicking on: **Target** | "**Recorder Configuration**" | **J-Link**.
    - Set-up the **Recorder Configuration** window as shown in *Figure A.1*:

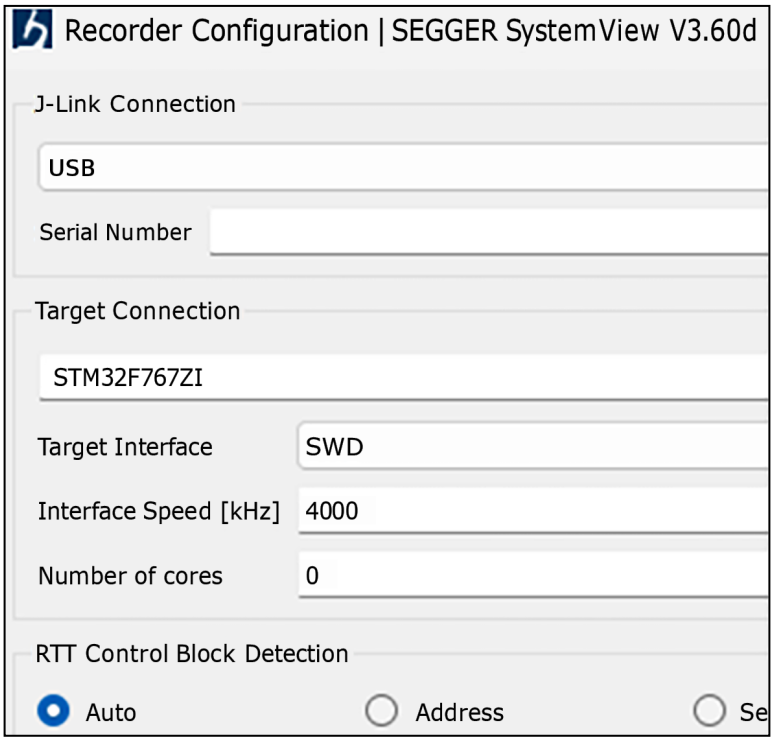


Figure A.1: SystemView's Recorder Configuration

3. Ensure the target program is running on the dev-board, e.g., via Ozone.
4. To avoid overflow with SystemView Recorder, stop the Ozone debug-session, if it is running. This can be done in Ozone by clicking on the red on/off icon, or by clicking on **Debug** | "**Stop Debug Session**".
5. When SystemView Recorder can be run:
    - After the target-program runs `SEGGER_SYSVIEW_Conf()`.
    - If FreeRTOS is being used, after the target-program runs `vTaskStartScheduler()`.
6. To run the SystemView Recorder:
    - On the toolbar, click on the play button (**Start Recording**).
7. To see the messages generated by `SEGGER_SYSVIEW_PrintfHost()`:
    - Click on the **Terminal** tab.
8. To make the timestamp format consistent among the windows:
    - On the toolbar, click on: **View** | "**Display recording time**".

Appendix B: Reference Information

This appendix lists the primary reference-documents for the hardware and the software-tools used in the book, such as user-guides and reference manuals. The primary webpages are also listed on the OEMs' websites.

This appendix is organized by chapter. Usually, a reference is only listed for the first chapter in which it is cited.

The book's primary tools are STM32CubeIDE, Ozone, and SystemView. For each tool, its help interface provides links to documentation that is specifically for the tool's version. This is especially helpful if you're not using the latest version of the tool. On the tool's website, it may be challenging to find documentation for prior versions.

How to search for additional documents:

- **Arm**: The search webpage is at: `https://developer.arm.com/`. Typically, a link is returned for the document's webpage. Also, that webpage has a link to the document's pdf version.
- **SEGGER**: In our experience, the best way to find documents on SEGGER's websites is by using an Internet search. For many documents, there are pdf and webpage versions.
- **ST**: The search webpage is at: `https://www.st.com/content/st_com/en/search.html`

If a listed document is not found on the OEM's website, it can potentially be found via an Internet search.

Chapters 2 and 3: Introducing the dev-board and tools

Chapters 2 and 3 present the hardware and software used in the book. This section lists the hardware and software's primary reference-documents and webpages.

Hardware documentation and webpages

The dev-board: STM32 Nucleo-F767ZI

Documentation:

- Data brief: *STM32 Nucleo-144 boards*. Document ID: DB3171
- User manual: *STM32 Nucleo-144 boards (MB1137)*. Document ID: UM1974
- User manual: *Getting started with STM32 Nucleo board software development tools*. Document ID: UM1727

Webpages:

- The dev-board:
    - *NUCLEO-F767ZI - STM32 Nucleo-144 development board with STM32F767ZI MCU*. `https://www.st.com/en/evaluation-tools/nucleo-f767zi.html`
- The related dev-board families:
    - *STM32 MCU & MPU Eval Tools*. `https://www.st.com/en/evaluation-tools/stm32-mcu-mpu-eval-tools.html`
    - *STM32 Nucleo Boards*. `https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html`

The MCU: STM32F767ZI

Documentation:

- Datasheet: *STM32F765xx, STM32F767xx, STM32F768Ax, STM32F769xx*. Document ID: DS11532
- Errata sheet: *STM32F76xxx/77xxx device errata*. Document ID: ES0334
- Example programs:
    - Application note: *STM32Cube MCU Package examples for STM32F7 Series*. Document ID: AN4731
    - Software: `https://github.com/STMicroelectronics/STM32CubeF7`
- Programming manual: *STM32F7 Series and STM32H7 Series Cortex-M7 processor programming manual*. Document ID: PM0253
- Reference manual: *STM32F76xxx and STM32F77xxx advanced Arm-based 32-bit MCUs*. Document ID: RM0410

Webpages:

- The MCU:
    - *STM32F767ZI*. `https://www.st.com/en/microcontrollers-microprocessors/stm32f767zi.html`
- The related MCU-families:
    - *STM32 32-bit Arm Cortex MCUs*. `https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html`
    - *STM32 High Performance Microcontrollers (MCUs)*. `https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html`
    - *STM32F7 - Very high-performance MCUs with Cortex-M7*. `https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html`
    - *STM32F7x7*. `https://www.st.com/en/microcontrollers-microprocessors/stm32f7x7.html`

Arm Cortex-M7

ST documentation:

- Product training: *STM32F7_System_Core*. Document ID: none (search for title)
- Programming manual: *STM32F7 Series and STM32H7 Series Cortex-M7 processor programming manual*. Document ID: PM0253
- Reference manual: *STM32F76xxx and STM32F77xxx advanced Arm-based 32-bit MCUs*. Document ID: RM0410

ST webpages:

- *Arm Cortex-M7 in a nutshell*. `https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers/arm-cortex-m7.html`

Arm documentation:

- Datasheet: *Arm Cortex-M7 Processor Datasheet*. Document ID: none (search for title)
- Reference manual: *Arm Cortex-M7 Processor Technical Reference Manual*. Document ID: ARM DDI 0489F
- User guide: *ARM Cortex-M7 Devices Generic User Guide*. Document ID: ARM DUI 0646C
- White Paper: *ARM Cortex-M for Beginners*. Document ID: none (search for title)

Arm webpages:

- Cortex-M7: `https://developer.arm.com/Processors/Cortex-M7`
- Cortex-M7 documentation search-page: `https://developer.arm.com/Processors/Cortex-M7#Resources`

ST-LINK/V2-1

Documentation:

- Data brief: *ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32 microcontrollers*. Document ID: DB1275
- Technical note: *Overview of ST-LINK derivatives*. Document ID: TN1235
- User manual: *STM32 Nucleo-144 boards (MB1137)*. Document ID: UM1974
- User manual: *ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32*. Document ID: UM1075

Webpages:

- *NUCLEO-F767ZI - STM32 Nucleo-144 development board with STM32F767ZI MCU*. `https://www.st.com/en/evaluation-tools/nucleo-f767zi.html`
- *ST-LINK/V2*. `https://www.st.com/en/development-tools/st-link-v2.html`

Software documentation and webpages

STM32CubeIDE

ST documentation:

- Installation guide: *STM32CubeIDE installation guide*. Document ID: UM2563
- User guide: *STM32CubeIDE user guide*. Document ID: UM2609
- Manuals for the tools included in STM32CubeIDE, e.g., build tools, GNU tool-chain, etc.
    - Available in STM32CubeIDE at **Help** : **Information Center**

ST webpages:

- *STM32CubeIDE : Integrated Development Environment for STM32*. `https://www.st.com/en/development-tools/stm32cubeide.html`
- ST Wiki: *STM32CubeIDE*. `https://wiki.stmicroelectronics.cn/stm32mcu/wiki/Category:STM32CubeIDE`

J-Link

ST Documentation:

- Data brief: *Segger J-Link debug and flash programming probes*. Document ID: 029233

SEGGER Documentation:

- User guide: *J-Link / J-Trace*. Document ID: UM08001

ST webpages:

- *Segger J-Link*. `https://www.st.com/en/partner-products-and-services/segger-j-link.html`

SEGGER webpages:

- *Converting ST-LINK On-Board Into a J-Link*. `https://www.segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board/`

Ozone

Documentation:

- Reference manual: *Ozone : User Guide & Reference Manual*. Document: UM08025
    - Available in Ozone at **Help** | **User Guide...**.

Webpages:

- *Ozone : Debugger and performance analyzer*. `https://www.segger.com/products/development-tools/ozone-j-link-debugger/`
- SEGGER Wiki: *Ozone*. `https://wiki.segger.com/Ozone`

SystemView

Documentation:

- User guide: *SystemView : User Guide*. Document UM08027
    - Available in SystemView at **Help** | **User Guide...**.

Webpages:

- *SystemView : The real-time software analysis tool*. `https://www.segger.com/products/development-tools/systemview/`
- SEGGER Wiki : *SystemView*. `https://wiki.segger.com/SystemView`

Chapter 4: Understanding super-loops

Chapter 4 presents a super-loop design.

The MCU: STM32F767ZI

Application note: *Managing low-power consumption on STM32F7 Series microcontrollers*. Document ID: AN4749

Chapter 5: Implementing the super-loop

Chapter 5 presents a super-loop example-program, and how to create it.

Hardware documentation and webpages

The MCU: STM32F767ZI

- Book: *Mastering STM32*, second edition. C. Noviello, 2022
- User manual: *Description of STM32F7 HAL and low-layer drivers*. Document ID: UM1905
- User guide: *STM32CubeMX for STM32 configuration and initialization, C code generation*. Document ID: UM1718

Arm Cortex-M7

- Application note: *ARM Cortex-M Programming Guide to Memory Barrier Instructions*. Document ID: Application Note 321
- Knowledge base article: *ARM: Cortex-M3/M4 Interrupts Happening Twice?*. Article ID: KA003795. (Information on memory barriers.). `https://developer.arm.com/documentation/ka003795/latest`
- Book: *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, third edition. J. Yiu. 2014
- Reference manual: *ARMv7-M Architecture Reference Manual*. Document ID: ARM DDI 0403E.b
- Reference manual: *CMSIS-Core (Cortex-M) : CMSIS-Core support for Cortex-M processor-based devices*. `https://arm-software.github.io/CMSIS_5/Core/html/index.html`
- Tutorial: *A Practical guide to ARM Cortex-M Exception Handling*. C. Coleman, 2019. `https://interrupt.memfault.com/blog/arm-cortex-m-exceptions-and-nvic`
- User guide: *Arm Cortex-M7 Devices Generic User Guide*. Document ID: ARM DUI 0646C

Software documentation and webpages

STM32CubeIDE

- User guide: *STM32CubeMX for STM32 configuration and initialization, C code generation*. Document ID: UM1718

Chapters 6 and 7: RTOS tasks and the FreeRTOS scheduler

Chapter 6 presents RTOS tasks and how to run and debug a FreeRTOS program. *Chapter 7* further describes the FreeRTOS tasks and the scheduler, as well as how to work with them.

FreeRTOS

Documents:

- FreeRTOS.org documents
    - These FreeRTOS.org documents are somewhat dated. The FreeRTOS website appears to have the most up-to-date reference information. `https://www.freertos.org/Documentation/02-Kernel/07-Books-and-manual/01-RTOS_book`
- Book: *Mastering the FreeRTOS™ Real Time Kernel : A Hands-On Tutorial Guide*. R. Barry. 2023. (includes example-program source code)
- Reference manual: *The FreeRTOS™ Reference Manual : API Functions and Configuration Options*. 2017

FreeRTOS website:

- Home page: `https://www.freertos.org/`
- Forums: `https://forums.freertos.org/`
- Reference information
    - FreeRTOS API Reference: `https://www.freertos.org/Documentation/02-Kernel/04-API-references/01-Task-creation/00-TaskHandle`
    - FreeRTOS configuration file (`FreeRTOSConfig.h`): `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/02-Customization`

- Topics
    - FreeRTOS naming conventions for functions and variables: https://www.freertos.org/Documentation/02-Kernel/06-Coding-guidelines/02-FreeRTOS-Coding-Standard-and-Style-Guide
    - FreeRTOS tick interrupt: https://www.freertos.org/Documentation/02-Kernel/05-RTOS-implementation-tutorial/02-Building-blocks/03-The-RTOS-tick
    - FreeRTOS features, including memory: https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/00-Developer-docs

Chapter 10: Drivers and ISRs

Chapter 10 describes creating drivers, and using ISRs and UARTs.

The MCU: STM32F767ZI

- Presentation slides: *STM32F7 - USART : Universal Synchronous/Asynchronous Receiver Transmitter Interface*. from ST. Document ID: none. https://www.st.com/resource/en/product_training/STM32F7_Peripheral_USART.pdf

FreeRTOS

- *Running the RTOS on a ARM Cortex-M Core*. (includes interrupt priorities) https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/ARM-Cortex/RTOS-Cortex-M3-M4

Chapter 11: More efficient drivers and ISRs

Chapter 11 describes creating drivers, and using DMA and stream-buffers

The MCU: STM32F767ZI

- Application notes: *Using the STM32F2, STM32F4 and STM32F7 Series DMA controller*. Document ID: AN4031.
- Presentation slides: *STM32F7 - DMA : Direct memory access controller*. from ST. Document ID: none. https://www.st.com/resource/en/product_training/STM32F7_System_DMA.pdf

FreeRTOS

- Stream-buffers tutorial: https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/04-Stream-and-message-buffers/02-Stream-buffer-example

Chapter 12: Sharing hardware peripherals among tasks

Chapter 12 describes creating drivers for use in writing data to a USB connection.

The following are sources on USB, from ST:

- USB tutorials:
    - Wiki: *Introduction to USB with STM32*. `https://wiki.st.com/stm32mcu/wiki/Introduction_to_USB_with_STM32`
    - Slides: *STM32F7 - USB : USB Full Speed and High Speed interfaces*. Revision 1.0. `https://www.st.com/resource/en/product_training/STM32F7_Peripheral_USB_OTG_FS_HS.pdf`
- USB course, *STM32 USB training*.
    - `https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-moocs/STM32-USB-training.html`
    - Videos: `https://www.youtube.com/playlist?list=PLnMKNibPkDnFFRBVD206EfnnHhQZI4Hxa`
    - Course materials (slides and code): `https://drive.google.com/file/d/1sjU9iNvh_khDZHDM9Qau03PGKwMd4u7U/view`
- USB example-programs for STM32F7 MCUs:
    - Application note : *STM32Cube MCU Package examples for STM32F7 Series*. Document ID: AN4731
- User and reference manuals:
    - User manual: *STM32Cube USB device library*. Document ID: UM1734
    - User manual: *STM32Cube USB host library*. Document ID: UM1720
    - User manual: *STM32CubeMX for STM32 configuration and initialization C code generation*. Document ID: UM1718
    - Reference manual: *STM32F76xxx and STM32F77xxx advanced Arm-based 32-bit MCUs*. Document ID: RM0410

Chapter 14: FreeRTOS memory management

Chapter 14 describes FreeRTOS memory management.

Heap and stacks

FreeRTOS.org

- *Stack Usage and Stack Overflow Checking*. `https://www.freertos.org/Stacks-and-stack-overflow-checking.html`
- *[Heap] Memory Management*. `https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/09-Memory-management/01-Memory-management`
- *Static Vs Dynamic Memory Allocation*. `https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/09-Memory-management/03-Static-vs-Dynamic-memory-allocation`
- Book: *Mastering the FreeRTOS™ Real Time Kernel : A Hands-On Tutorial Guide*, *Chapter 3 Heap Memory Management*

ST

- User guide: *STM32CubeIDE user guide*. Document ID: UM2609
    - 2.2.1.6 *Linker script*. Table 3 “Memory map layout”
    - 2.4 *Build Analyzer*. (tool, shows memory layout for ELF)
    - 2.6 *Static Stack Analyzer*. (tool, shows stacks for ELF)

Memory protection unit (MPU)

FreeRTOS.org

- *Memory Protection Unit (MPU) Support*. `https://www.freertos.org/Security/04-FreeRTOS-MPU-memory-protection-unit`
- *Benefits of Using the Memory Protection Unit*. `https://www.freertos.org/Community/Blogs/2021/benefits-of-using-the-memory-protection-unit`
- *Chapter 7 : FreeRTOS-MPU*. `https://en.freertos.org/media/MPU_Chapter.pdf`
- *FreeRTOS-MPU Specific Functions*. `https://freertos.org/Documentation/02-Kernel/04-API-references/13-FreeRTOS-MPU-specific/00-FreeRTOS-MPU-specific`

ST

- Application note: *Introduction to memory protection unit management on STM32 MCUs*. Document ID: AN4838
- Knowledge base article: *How to configure the MPU of an STM32 using STM32CubeMX*. `https://community.st.com/t5/stm32-mcus/how-to-configure-the-mpu-of-an-stm32-using-stm32cubemx/ta-p/49825`
- Application note: *STM32Cube MCU Package examples for STM32F7 Series*. Document ID: AN4731. Example: *FreeRTOS_MPU*
- Programming manual: *STM32F7 Series . . . Cortex-M7 processor programming manual*. Document ID: PM0253, Section 4.6 *Memory protection unit*

Appendix C: FreeRTOS installation

Appendix C describes creating FreeRTOS-projects, and installing FreeRTOS. The references provided here are for installing FreeRTOS.

FreeRTOS.org, documents

- Book: *Mastering the FreeRTOS™ Real Time Kernel : A Hands-On Tutorial Guide*. R. Barry, 2023. *Chapter 2*, *The FreeRTOS Kernel Distribution*
- Reference manual: *The FreeRTOS™ Reference Manual : API Functions and Configuration Options*. 2017. *Chapter 7*, *Kernel Configuration* (`FreeRTOSConfig.h`)
- Webpage: *Customization* (`FreeRTOSConfig.h`). `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/02-Customization`

FreeRTOS.org, installation guidance

The webpages referenced here are from FreeRTOS.org. FreeRTOS supports a variety of MCUs. These webpages are related to installing FreeRTOS, for the book's dev-board and MCU. For readers who want to know more about FreeRTOS installation, these webpages can be a good place to start.

- *Build your first FreeRTOS project*. `https://www.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/03-Build-your-first-project`

The above webpage states, "it is always recommended that a new FreeRTOS project [be] created by starting with, and then adapting, one of the provided preconfigured demos." The preconfigured demos are described on this webpage:

- *FreeRTOS Demo Applications*. `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/01-Demo-overview`

In *Appendix C*, the FreeRTOS installation was not adapted from a preconfigured demo. It was adapted from the FreeRTOS installation generated by STM32CubeIDE. This included the `FreeRTOSConfig.h` file generated by the IDE. Also, the IDE selected the installation's compiler-specific source-files and architecture-specific source-files. Those source-files are at `FreeRTOS-Kernel\portable\GCC\ARM_CM7\r0p1`.

The preconfigured demo for our dev-board's MCU is described on the webpage below. This webpage can be useful for understanding the FreeRTOS installation presented in *Appendix C*.

- *Running the RTOS on an ARM Cortex-M Core*. `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/ARM-Cortex/RTOS-Cortex-M3-M4`

The next webpage presents a preconfigured demo for the STM32756G-EVAL Evaluation Kit. Its MCU is an STM32F7 ARM Cortex-M7. This webpage may be a useful reference, as the dev-board and MCU are similar to ours (though not identical). The demo uses IDEs, but not the STM32CubeIDE. The demo is available in the "classic" FreeRTOS distribution. It's not distributed with the FreeRTOS LTS package used in *Appendix C*.

- *ST ARM Cortex-M7 STM32 F7 RTOS Demo*. `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/ST-Microelectronics/ST_STM32F7_Cortex-M7_RTOS_Demo`

Information on how to modify a preconfigured demo for different hardware and IDEs is provided here:

- *Modifying a FreeRTOS Demo to use a different compiler or run on different hardware*. `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/04-Modifying-demos`

The webpage below is for installing FreeRTOS on a system for which there is no port or preconfigured demo. The webpage states, "You do not need to read or understand this page if using one of the many existing ports and demo applications!". So, the webpage isn't needed for installing FreeRTOS on our dev-board, since a demo is available for its MCU.

- *Creating a New FreeRTOS Port*. `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/01-FreeRTOS-porting-guide`

ST

These ST sources are on the FreeRTOS installation. They are the most applicable ST sources we could find, for the book's dev-board and IDE.

This is a course from ST, on CMSIS-RTOS, using FreeRTOS. A NUCLEO-L476RG dev-board is used, with a Cortex-M4-based STM32 MCU. There are slides on hardware-configuration for FreeRTOS, using STM32CubeIDE (pages 11 to 38, 81 to 84, and 120 to 124):

- Course: *FreeRTOS on STM32 CMSIS_OS API v2.0*
    - Webpage: *FreeRTOS on STM32 v2 MOOC*. `https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-moocs/freertos-common-microcontroller-software-interface-standard-osv2.html`
    - Slides: *FreeRTOS on STM32 v2 - materials.zip*. `https://drive.google.com/drive/folders/1yALzIl_zjftUID3aAWHSE3cJWfmixsSO`

The user guide, below, is for STM32CubeMX, which is part of STM32CubeIDE. The user guide has two sections related to generating code for a FreeRTOS installation.

- User guide: *STM32CubeMX for STM32 configuration and initialization, C code generation*. Document ID: UM1718
    - Section 4.5.15, *FreeRTOS configuration panel*
    - Section B.3.5, *FreeRTOS.* (code-generation design choices)

Appendix C: Creating FreeRTOS Projects, and Installing FreeRTOS

To create your own FreeRTOS-based system, a typical way to start is with an IDE-project with FreeRTOS installed. (We'll refer to such an IDE-project as a *FreeRTOS-project*.) Also, in reading this book, the best way to learn the material is to augment study with hands-on experiments and development, and that also requires a FreeRTOS-project.

This appendix shows several techniques for creating a FreeRTOS-project. The techniques are presented in order of difficulty, starting with the least difficult.

The following topics are presented:

- Creating IDE-projects for use in learning FreeRTOS
- Copying and renaming projects (cloning)
- Installing FreeRTOS and SystemView

Several of the techniques for creating FreeRTOS-projects were introduced in chapter 6, in the section *Installing and configuring FreeRTOS*.

This appendix assumes you are familiar with the basics of FreeRTOS, as presented in *Chapter 6*, *Understanding RTOS Tasks*, and *Chapter 7*, *Running the FreeRTOS Scheduler*. It is also assumed that you are familiar with the IDE and debugging-tools used in the book. They are presented in *Chapter 3*, *Introducing the Development Tools*, and in *Chapter 5*, *Implementing the Super-Loop*.

Creating IDE-projects for use in learning FreeRTOS

When reading the book, learning can be greatly enhanced by writing prototype programs that use the FreeRTOS APIs. Also, modifying the book's example-programs can be instructive.

To write FreeRTOS applications, an IDE-project is needed, with FreeRTOS installed. There are two easy ways to obtain FreeRTOS projects:

Technique #1: Copy the example-programs' IDE-workspace

To make modifications to the book's example-programs, the example-program's IDE-workspace can be copied. The workspace has an IDE-project for each chapter with example-programs.

The workspace and its directory are named `packtBookRTOS2E` (described in chapter 3). The workspace also has folders for common code, used by the IDE-projects. For example, the directory `Middleware` has source-code for FreeRTOS and SystemView, and the directory is shared by most of the IDE-projects. (A few of the IDE-projects do not use this shared directory, e.g., `Chapter_03`.)

When the workspace directory is copied, the new directory can be renamed. The new workspace can be opened from STM32CubeIDE, e.g., **File | Switch Workspace | Other**.

Technique #2: Copy the FreeRTOS project that is provided in the book's code-repository

A FreeRTOS project is provided in the book's code-repository, named **FreeRTOS_project**. It has FreeRTOS and SystemView installed. The project can be copied, for use in creating prototype programs. However, copying an IDE-project is more involved than copying the project-directory and renaming it. The next section describes how to copy and rename projects, including FreeRTOS_project.

`FreeRTOS_project` is self-contained. It does not use the folders for common-code in `packtBookRTOS2E`. `FreeRTOS_project` is in the workspace-directory `packtBookRTOS2E`, from when the example-programs were installed.

`FreeRTOS_project` was created by using the instructions for installing FreeRTOS and SystemView, which are presented later in this appendix. In `FreeRTOS_project`, code was added to `main.c`, in `main()`. The code was added in USER CODE blocks. In this added code, FreeRTOS APIs are called to create tasks and start the scheduler. Also, the tasks are defined in `main.c`.

Copying and renaming projects (cloning)

This section shows how an STM32CubeIDE project can be copied and renamed (i.e., *cloned*). It involves several steps, and it's not as simple as just copying the project-directory and renaming it.

The project that is copied will be referred to as the "source-project." When the source-project is copied, the project that is created will be referred to as the "clone-project."

Creating a FreeRTOS-project from scratch is time-consuming. Rather than create a FreeRTOS-project every time one is needed, we can create a FreeRTOS-project and save it, then copy it to create new FreeRTOS-projects. Typically, we also need to rename these newly created projects.

Of course, there are other reasons for renaming a project, besides cloning. The instructions here for renaming a project can be applied to any IDE-project, not just one that has been copied.

As far as we know, ST does not provide instructions for copying and renaming projects. Also, STM32CubeIDE (v1.7.0) has a Rename feature for projects, but it appears the feature does not make all of the updates needed for renaming.

The instructions provided here are based on our research, and they have worked for us. The instructions assume that the source-project and clone-project are on the same development computer, e.g., the same STM32CubeIDE instance is used. Though, if the source and clone projects are on different development-computers, the copying and renaming process will likely work. Also, it's assumed that the source and clone projects are used on the same target hardware, e.g., used on the same model dev-board.

For a source-project to be copied, its configuration must be amenable to copying. For example, its build-configuration should specify file-paths that are relative to the current project and workspace, and not use absolute paths.

In the instructions below, we'll show how to copy and rename an IDE-project, in general, and we'll illustrate the process with a specific example. In the example, the source-project is `FreeRTOS_project`, which is provided in the book's code-repository (described earlier). And, the clone-project will be named `FreeRTOS_experiment_01`.

Copy the source-project's directory

First, we'll copy the source-project's directory, and paste it to create the clone-project's directory:

- In Windows File Explorer, navigate to the source-project's workspace directory. In our example, the workspace-directory is `packtBookRTOS2E` (described earlier). It contains the projects from the book's code-repository, including the project `FreeRTOS_project`.
- Copy the source-project's directory. In our example, it is `FreeRTOS_project`.
- Then, paste the copied-directory into the desired directory for the clone-project. We'll paste it into the directory `C:\projects\packtBookRTOS2E_experiments`, which we'll use as a workspace-directory.
- Rename the pasted directory. It is the clone-project's directory. We'll rename the directory to `FreeRTOS_experiment_01`. The directory's full-path is `C:\projects\packtBookRTOS2E_experiments\FreeRTOS_experiment_01`

Delete unnecessary files

Next, we'll delete the unnecessary files and folders in the clone-project directory. The following files and folders should be deleted, if they are present:

- The directory `.settings`. (It has the user's configuration-settings for the project. Do not delete it if the source-project's settings should be preserved.)
- The directories `Debug` and `Release`. (They will be re-created during build.)
- The file `.mxproject`. (It records the files created for code generation. Full paths are specified for the files, so `.mxproject` isn't portable to another computer. The file `.mxproject` will be recreated if code is generated.)
- The file `*.jdebug.user`. (It is the Ozone user-file for the project. It will be re-created when Ozone is used.)

Rename the project

In the clone-project, we'll need to change any instances of the source-project's name to the clone-project's name. For the example, we'll change instances of `FreeRTOS_project` to `FreeRTOS_experiment_01`.

- The `*.ioc` file's name contains the name of the source-project. Change it to the clone-project's name. We changed the filename from `FreeRTOS_project.ioc` to `FreeRTOS_experiment_01.ioc`.
- There are configuration files whose contents specify the source-project's name. Those files are listed below. In those files, use a text editor to change the instances of the source-project's name to the clone-project's name. We used Notepad++ to change instances of `FreeRTOS_project` to `FreeRTOS_experiment_01`.
    - `.cproject`
    - `.project`
    - The `*.ioc` file. (In our example, it is `FreeRTOS_experiment_01.ioc`.)

If there is an Ozone project-file (`*.jdebug`), it specifies the path to the executable file. That path will need to be updated.

The clone-project is now ready for use. STM32CubeIDE can be used to import the clone-project into a workspace. How to import projects is described in *Chapter 3*, *Introducing the Development Tools*. When using the clone-project, if the `*.ioc` file will be edited, and code generated, it may be prudent to first make a backup-copy of the project.

We have now seen how an existing FreeRTOS project can be cloned. Next, we'll see how to create a FreeRTOS project from scratch.

Installing FreeRTOS and SystemView

This section provides instructions for installing FreeRTOS and SystemView in an IDE-project.

The instructions are presented in the following subsections:

- Overview
- The installation process
- Creating the IDE-project
- Installing FreeRTOS
- Writing and running a FreeRTOS program
- Installing SystemView

Overview

We'll create a FreeRTOS project and install the current version of FreeRTOS (current as of the time of writing this appendix). This requires manually installing FreeRTOS. The present section explains why we're manually installing FreeRTOS, and not using the IDE to generate the FreeRTOS installation. In addition, an overview is provided for our process of installing FreeRTOS.

The easiest way to create a FreeRTOS project is to use STM32CubeIDE to create the project and generate the FreeRTOS installation. However, that FreeRTOS installation has significant limitations. The biggest limitation is that the FreeRTOS version can be dated, or at least not be the current version. So, the approach we'll take is to use the IDE to generate part of the project, and we'll manually install the current version of FreeRTOS.

Even if using an older FreeRTOS version was acceptable, there are two other limitations in using FreeRTOS-installations generated by the IDE. These specific limitations depend on whether the IDE is configured to generate `main()`, in `main.c`. This configuration-setting is specified in the `*.ioc` file's **Project Manager** window, in the checkbox **Do not generate the main().**

If `main()` is generated in `main.c`, the FreeRTOS APIs are not intended to be called directly. Instead, the project's programs are expected to call CMSIS-RTOS wrapper-functions that call FreeRTOS APIs. The purpose of CMSIS-RTOS is to provide a generic RTOS interface, which can be implemented by a variety of RTOSes, such as FreeRTOS. Applications that use CMSIS-RTOS can be easily ported between systems with different RTOSes.

For FreeRTOS-installations generated by the IDE, the generated `main()` has calls to CMSIS-RTOS APIs. Those API calls are used to create a task, and to start the scheduler to run the task. However, those API calls are not in `USER CODE` sections. The API-calls can be removed manually, by editing `main.c`. However, those API calls will be re-generated in `main.c` whenever the `*.ioc` file is changed, and code is re-generated.

If `main()` is not generated in `main.c`, then presumably, the user will need to create `main()`. And, this `main()` will need to call the MCU set-up functions that the IDE generates in `main.c`, such as `SystemClock_Config()`. Though, we have not implemented this approach. Also, we could not find documentation for the IDE's configuration-setting **Do not generate the main()**.

There is yet another limitation of using the IDE to generate the FreeRTOS installation. The installation gets overwritten whenever system-code is re-generated, e.g., when the system configuration-file (`*.ioc`) is changed. Any user modifications to the FreeRTOS source-code would then be overwritten. For instance, using SystemView with FreeRTOS can require patching the FreeRTOS source-code, and those changes would be overwritten.

So, by manually installing FreeRTOS, we can use its current version. And, we'll avoid the problems that are encountered when using the IDE to generate the FreeRTOS installation.

The sections below show how to create a FreeRTOS project. In our example, it is named `FreeRTOS_project`, but any other name can be used. The IDE will be used to create the IDE-project, and to configure the hardware to run FreeRTOS. The IDE will also be used to generate the FreeRTOS configuration file, `FreeRTOSConfig.h`. This will ensure the configuration file is setup appropriately for our hardware, and that general-purpose configuration options are used, which are suitable for experimenting with FreeRTOS. The FreeRTOS source-code will then be installed manually, and the build-system will be configured for the project.

The `FreeRTOSConfig.h` generated by the IDE is for the FreeRTOS version that the IDE generates. The FreeRTOS we'll install is a newer version. FreeRTOS typically provides backward compatibility, so we're assuming the generated `FreeRTOSConfig.h` will be compatible with the installed FreeRTOS.

If you encounter problems in creating `FreeRTOS_project`, you can compare your project with `FreeRTOS_project` from the book's code-repository (it was described earlier). That project was created using the instructions on installing FreeRTOS, provided in the present section. SystemView is also installed in that project, and it was installed using the instructions that are presented later, in the section *Installing SystemView*.

When adding files to the project `FreeRTOS_project`, if the files are not visible in the IDE, use the IDE's Refresh function (in Project Explorer, click on the project, then click on **File | Refresh**).

The installation process

This section describes the FreeRTOS installation process, as a whole. The step-by-step installation process starts in the next section.

FreeRTOS installation is non-trivial, especially considering the many MCUs that it supports. The FreeRTOS configuration file, `FreeRTOSConfig.h`, has about 80 configuration parameters. Also, system interrupts must be configured properly, timers configured, etc. And, the FreeRTOS kernel has code files that must be refactored for the MCU and compiler that is used. Indeed, a book could be written just on FreeRTOS installation.

The FreeRTOS installation-process that is presented here has been simplified, to avoid much of those complexities. Some of the simplifications are:

- The installation process is for our dev-board and does not address installation on other hardware.
- The installed FreeRTOS is intended for experimenting with FreeRTOS, in conjunction with reading the book. So a relatively simple set of general-purpose FreeRTOS features are used.
- STM32CubeIDE is used to configure the hardware for FreeRTOS use, and to generate the code needed for that. Most of the IDE's default hardware-settings are appropriate for using FreeRTOS on our MCU. A small number of configuration changes are needed.
- STM32CubeIDE is also used to configure and generate `FreeRTOSConfig.h`. Most of the IDE's default FreeRTOS-settings should be appropriate for our MCU, and for general-purpose FreeRTOS use. A small number of changes are needed for `FreeRTOSConfig.h`.
- The hardware configuration and code-generation is specified in STM32CubeIDE. It is taken mostly from ST tutorials on installing FreeRTOS. This helps ensure that the configuration is correct for our hardware, and appropriate for general-purpose use.

In presenting the installation process, the steps are specified in detail. However, there's little description of why particular configuration-settings were used. Unfortunately, that was beyond the scope of the book.

Sources of information on FreeRTOS installation are limited. FreeRTOS.org provides some very good information, but using it requires a high degree of skill, and a substantial amount of research (see *Appendix B*, *Reference Information*).

To learn about FreeRTOS installation, we suggest using the installation-process provided here, as an introduction. Individual configuration settings can be investigated. This can provide a conceptual foundation for learning about FreeRTOS installation, in general.

The sources on FreeRTOS installation that we found most helpful are presented in *Appendix B*, *Reference Information*.

Creating the IDE-project

First, create the initial IDE project, and configure the hardware, as described in *Chapter 5*, *Implementing the Super-Loop*. Those instructions include enabling the RNG (random number generator). However, enabling the RNG is optional here. For the present example, we'll use these settings:

- Workspace: `C:\projects\packtBookRTOS2E_experiments`
- Project name: `FreeRTOS_project`

For the `FreeRTOS_project` provided in the book's code-repository, the RNG is not enabled.

For the workspace, the IDE's editor can be configured as the default for Ozone configuration files (`*.jdebug`). This is described in *Chapter 3*, *Introducing the Development Tools*.

Installing FreeRTOS

Now that we're familiar with the installation process as a whole, let's look at the actual steps for installing FreeRTOS, as presented in the following sections:

- Configure the hardware to run FreeRTOS
- Use the IDE to create FreeRTOSConfig.h
- Manually install the FreeRTOS source-code
- Configure the build-system for FreeRTOS

Configure the hardware to run FreeRTOS

Next, we'll use the IDE to configure the hardware to run FreeRTOS.

- In the IDE, open `FreeRTOS_project.ioc`.
- Click on the tab **Pinout & Configuration**, then the tab **A -> Z**.
- Under the tab, the component to be configured can be selected.

Under the tab **A -> Z**, select **SYS**, for the system configuration-settings.

- In the drop-down box **Debug**, select **Serial Wire**. (This appears to be the default, even if **Disable** is selected.)
- In the drop-down box **Timebase Source**, select **TIM6**.
- This setting specifies the clock that is the HAL timebase. By default, HAL uses the SysTick timer. However, FreeRTOS also uses the SysTick timer for its timebase, by default. The STM32CubeMX user-manual has more information.

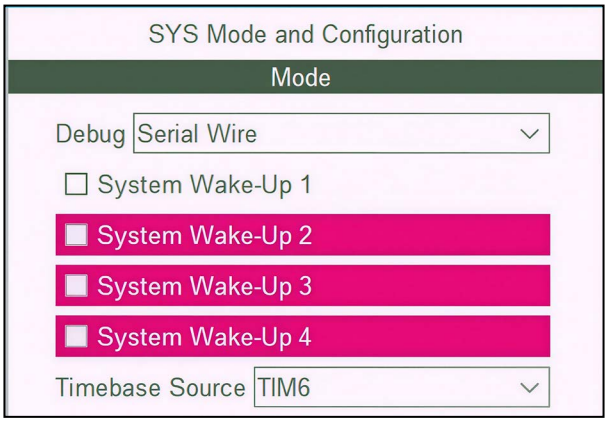


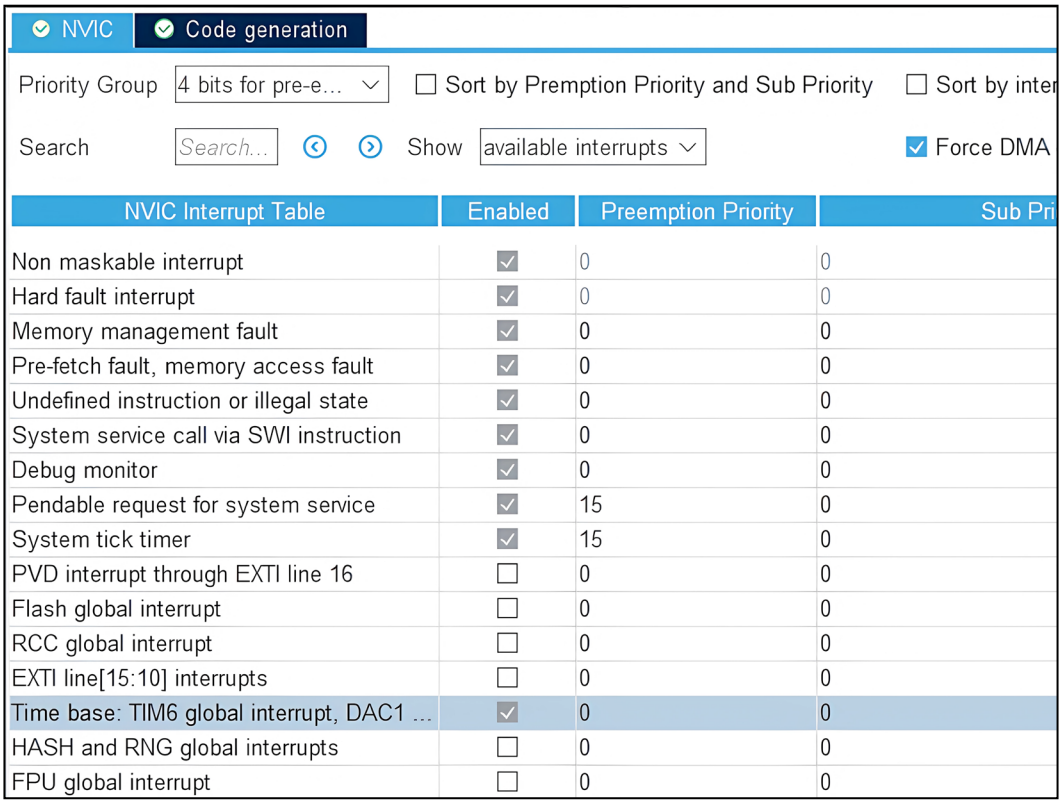
Figure C.1: Setting the debug interface and timebase

Under the tab **A -> Z**, select the **NVIC** configuration settings.

Configure the NVIC tab as shown in the figure below, including:

- In the drop-down box **Priority Group**, select **4 bits for pre-emption. . .**
- In the **NVIC Interrupt Table**, confirm these interrupts are enabled:
    - **System service call via SWI instruction**: Preemption Priority 0
    - **Pendable request for system service**: Preemption Priority 15
    - **System tick timer**: Preemption Priority 15

- **Time base: TIM6 global interrupt, ...** : Preemption Priority 0



NVIC Interrupt Table	Enabled	Preemption Priority	Sub Pri
Non maskable interrupt	☑	0	0
Hard fault interrupt	☑	0	0
Memory management fault	☑	0	0
Pre-fetch fault, memory access fault	☑	0	0
Undefined instruction or illegal state	☑	0	0
System service call via SWI instruction	☑	0	0
Debug monitor	☑	0	0
Pendable request for system service	☑	15	0
System tick timer	☑	15	0
PVD interrupt through EXTI line 16	☐	0	0
Flash global interrupt	☐	0	0
RCC global interrupt	☐	0	0
EXTI line[15:10] interrupts	☐	0	0
Time base: TIM6 global interrupt, DAC1 ...	☑	0	0
HASH and RNG global interrupts	☐	0	0
FPU global interrupt	☐	0	0

Figure C.2: NVIC configuration

Configure the **Code generation** tab as shown in *Figure C.3*, including:

- For the column **Generate IRQ handler**, the column header should be checked (on)
- Under **Generate IRQ handler**, ensure these settings are **not** checked:
  - **System service call via SWI instruction**
  - **Pendable request for system service**
  - **System tick timer**

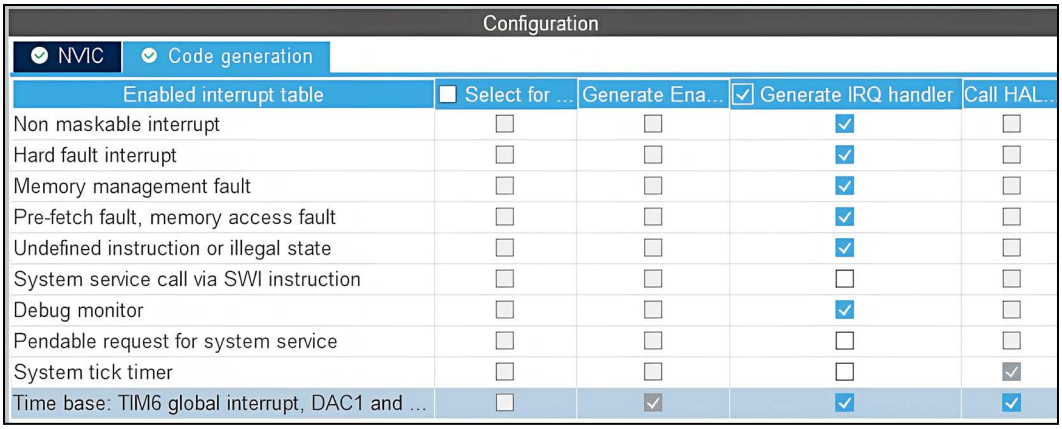


Figure C.3: NVIC code-generation settings

Having configured the hardware settings for FreeRTOS, save the `*.ioc` file (`Ctrl + S`). Code will be generated.

Use the IDE to create FreeRTOSConfig.h

We can use the IDE to generate `FreeRTOSConfig.h`. The dev-board is intended for experimenting with the MCU. So, the IDE's FreeRTOS install is presumably configured for general-purpose use on the dev-board, including the `FreeRTOSConfig.h` that is generated.

We'll use the IDE to install FreeRTOS and generate `FreeRTOSConfig.h`. This `FreeRTOSConfig.h` will be copied for later use, for when we manually install the current FreeRTOS source-code. After copying `FreeRTOSConfig.h`, we will uninstall the FreeRTOS that was installed by the IDE.

Configure FreeRTOSConfig.h

First, we'll configure the FreeRTOS that will be generated by the IDE.

Open the IDE's FreeRTOS configuration-interface:

- In the IDE, open `FreeRTOS_project.ioc`.
- Click on the tab **Pinout & Configuration**, then the tab **A -> Z**.
- In the drop-down box, select **FreeRTOS**

- In the pane **FREERTOS Mode and Configuration**, select the Interface, **CMSIS_V2**

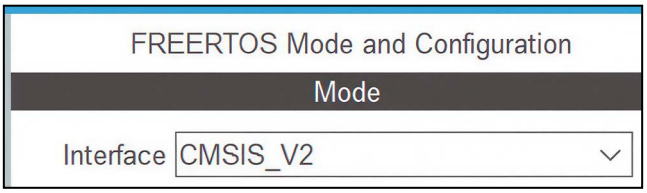


Figure C.4: Selecting the CMSIS-RTOS interface

In the FreeRTOS configuration pane, select the tab **Config parameters** (shown in *Figure C.5*). These settings specify the parameters used in the generated `FreeRTOSConfig.h`. Configure the settings as shown in *Figure C.5* to *Figure C.8*, below.

For each setting, when clicking on it, a description is provided at the bottom of the configuration pane. Detailed descriptions can be found at `FreeRTOS.org`, in the web-page for `FreeRTOSConfig.h`.

In *Figure C.5* to *Figure C.8*, most of the settings shown are default-settings. For the parameters in the bulleted-list below, ensure the parameters are set as specified:

- Under **MPU/FPU**, set **ENABLE_FPU** to **Enabled**:

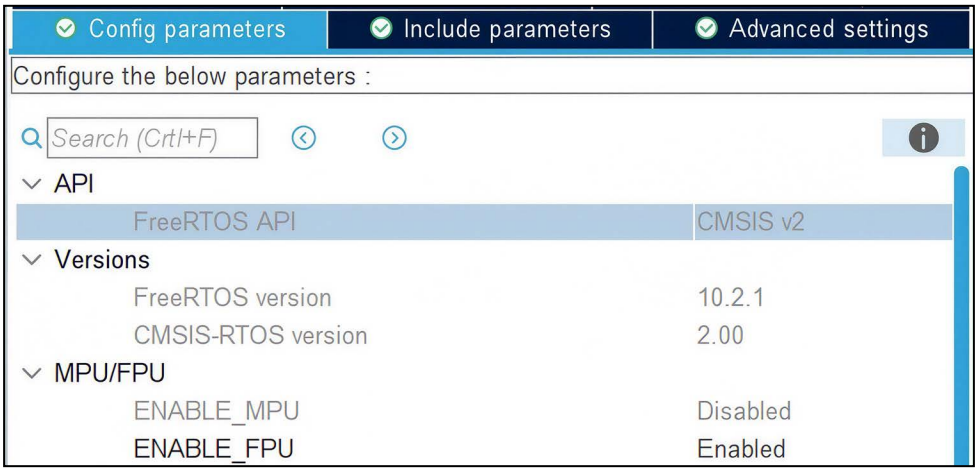


Figure C.5: Configuring FreeRTOSConfig.h, part 1

- Under **Kernel settings**, set **USE_PREEMPTION** to **Enabled**:

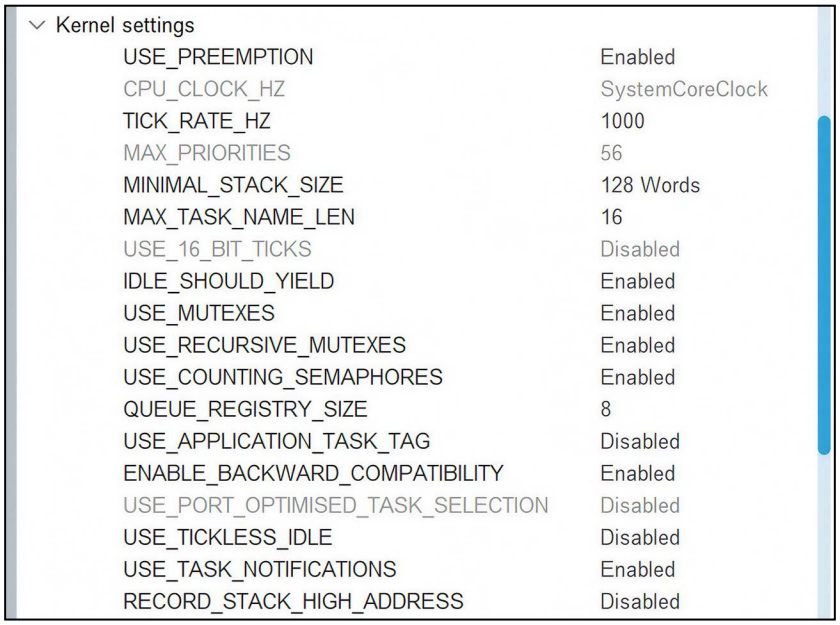


Figure C.6: Configuring FreeRTOSConfig.h, part 2

- Under **Memory management settings**, set **Memory Allocation** to **Dynamic / Static** and **Memory Management scheme** to **heap_4**:

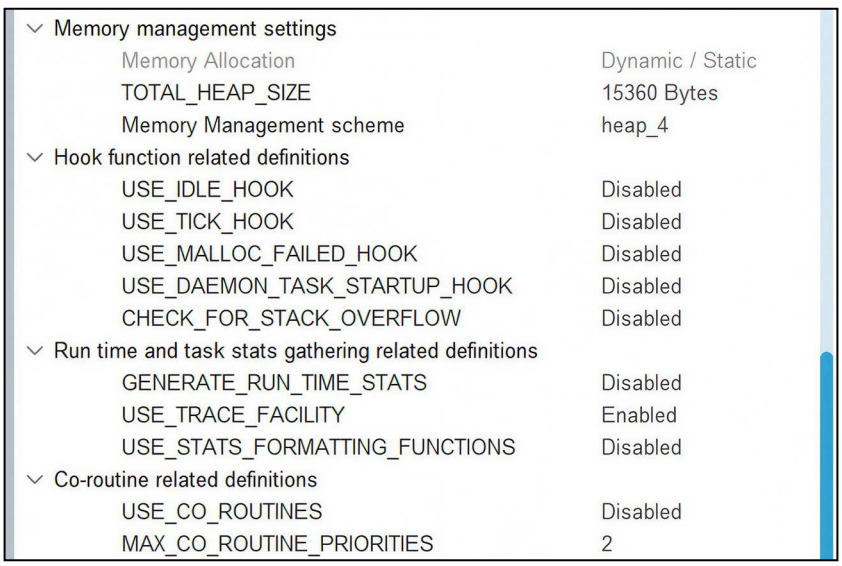


Figure C.7: Configuring FreeRTOSConfig.h, part 3

These configuration parameters use the following default settings.

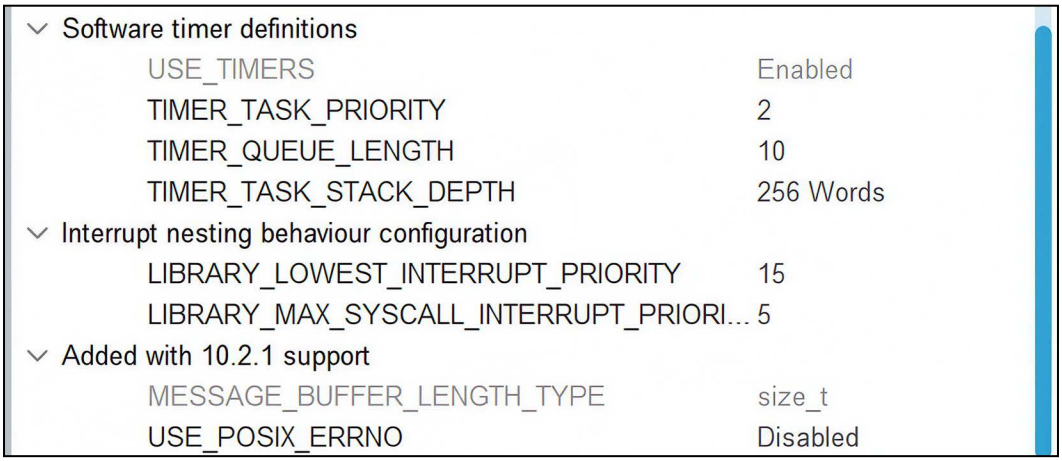


Figure C.8: Configuring FreeRTOSConfig.h, part 4

In the FreeRTOS configuration pane, select the tab **Advanced settings**. Configure the tab's pane as shown below. This setting may need to be set explicitly:

- Set **USE_NEWLIB_REENTRANT** to **Enabled**

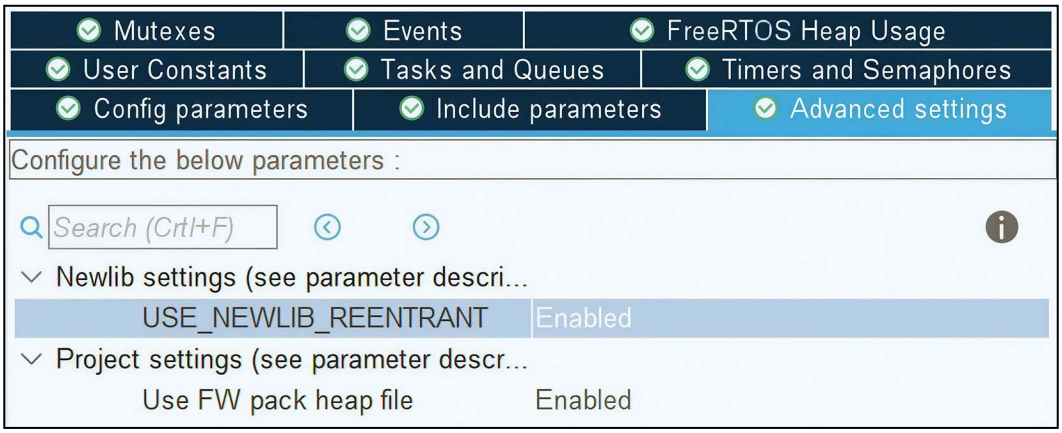


Figure C.9: The FreeRTOS "Advanced settings"

Generate FreeRTOS

Having configured FreeRTOS, we can now generate the code for it in the project.

- To generate the code, save the `*.ioc` file (`Ctrl + S`).

FreeRTOS will now be installed at: `Middlewares/Third_Party/FreeRTOS`. Part of that directory is shown in the following screenshot.

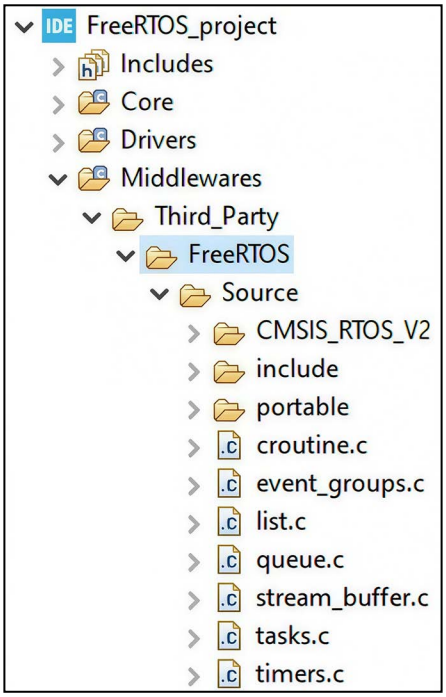


Figure C.10: The generated FreeRTOS source-code

Also, code is generated in `main.c` that creates a task and starts the scheduler (`Core/Src/main.c`). CMSIS-RTOS APIs are used, such as:

```
defaultTaskHandle = osThreadNew(StartDefaultTask, NULL,
                                &defaultTask_attributes);
. . .
osKernelStart();
```

Copy FreeRTOSConfig.h

The purpose of generating FreeRTOS was to create `FreeRTOSConfig.h`.

We'll copy that file into a directory. The directory and file will be used later, when we manually install FreeRTOS.

- At the project's root (`/FreeRTOS_project`), create the directories on this path: `Config/FreeRTOS/Inc`

- Copy the `FreeRTOSConfig.h` into that directory. Copy the file from `Core/Inc/FreeRTOSConfig.h`

Project Explorer should show this:

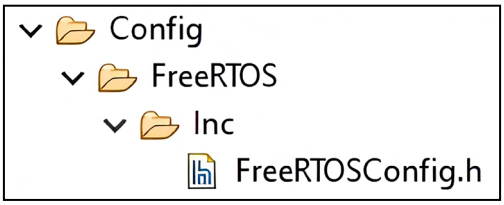


Figure C.11: Copying FreeRTOSConfig.h

Uninstall FreeRTOS

Now that we have copied the generated `FreeRTOSConfig.h`, we can uninstall the generated FreeRTOS:

- In the IDE, open `FreeRTOS_project.ioc`.
- Open the FreeRTOS configuration pane.
- In the **Interface** drop-down box, select **Disable** (shown below).

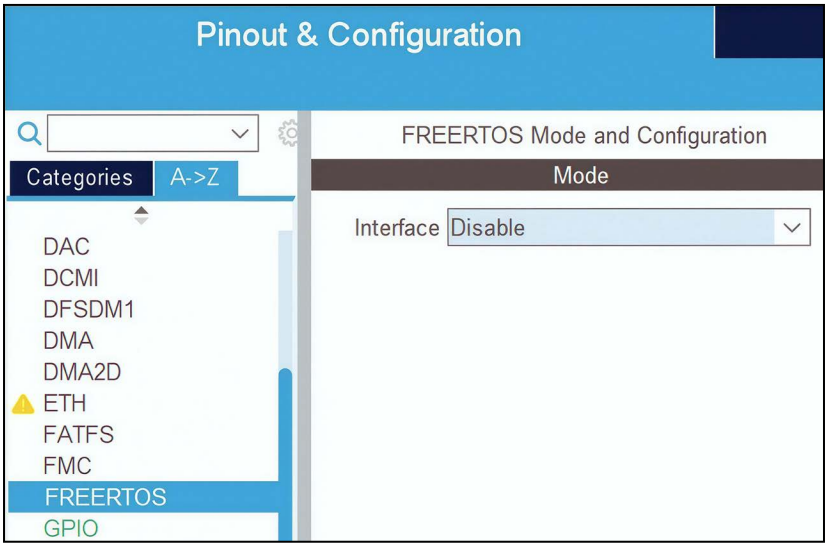


Figure C.12: Uninstalling the generated FreeRTOS source-code

- Save `FreeRTOS_project.ioc`, which will generate code for the new configuration settings.

- The FreeRTOS code that had been generated will be removed:
    - `Core\Inc\FreeRTOSConfig.h` is deleted.
    - In `main.c`, the calls to the `CMSIS-RTOS APIs` are removed.
    - The `Middlewares` directory is deleted, along with the directories under it.
    - If the `Middlewares` directory is not deleted, then manually delete the CMSIS-RTOS directory, if it exists. We won't need it: `Middlewares\Third_Party\FreeRTOS\Source\CMSIS_RTOS_V2`

Manually install the FreeRTOS source-code

At this point, the hardware has been configured to run FreeRTOS. Also, we have a `FreeRTOSConfig.h` file that is configured for the dev-board, and for general-purpose FreeRTOS use.

In this step, we'll manually install the FreeRTOS source-code. In addition, we'll create a program file with call-back functions needed for our FreeRTOS configuration.

Download the FreeRTOS source-code from `https://www.freertos.org/Documentation/02-Kernel/01-About-the-FreeRTOS-kernel/03-Download-freeRTOS/01-DownloadFreeRTOS`.

In this example, we're using FreeRTOS 202406.01 LTS (Long Term Support). LTS builds are for systems that will be supported long into the future. For programmers who prefer the latest features and bug fixes, the most recent version is available on GitHub, in the latest commit.

Expand the downloaded zip-file. The source-code that we'll copy is in:

```
FreeRTOSv202406.01-LTS\FreeRTOS-LTS\FreeRTOS\FreeRTOS-Kernel
```

In the FreeRTOS source-code, the kernel version is specified in: `FreeRTOS-Kernel\include\task.h`. For our download, the version is 11.1.0:

```
#define tskKERNEL_VERSION_NUMBER       "V11.1.0"
```

We will copy the FreeRTOS source-code into the project at:

```
Middlewares\Third_Party\FreeRTOS\Source
```

For the instructions below, you'll need to create the specified directories. Use Project Explorer, right-click, and select **New**.

- Install the kernel's program files:
    - From the directory `FreeRTOS-Kernel`, copy all of the files, but not the directories. Examples of the files are `queue.c` and `tasks.c`.

    - Paste the files into the project, in this directory: `Middlewares\Third_Party\FreeRTOS\Source`.
- Install the kernel's header files:
    - From the directory `FreeRTOS-Kernel\include`, copy all of the files. (Examples of the files are `queue.h` and `task.h`.)
    - Paste the files into the project, in this directory: `Middlewares\Third_Party\FreeRTOS\Source\include`.
- Install the kernel-files that are specific to our IDE's compiler and to our MCU:
    - From the directory `FreeRTOS-Kernel\portable\GCC\ARM_CM7\r0p1`, copy all of the files. (Examples of the files are `port.c` and `portmacro.h`.)
    - Paste the files into the project, in this directory `Middlewares\Third_Party\FreeRTOS\Source\portable\GCC\ARM_CM7\r0p1`.
- Install the memory-allocation implementation that will be used by the kernel:
    - Copy the file `FreeRTOS-Kernel\portable\MemMang\heap_4.c`
    - Paste the file into the project, in this directory: `Middlewares\Third_Party\FreeRTOS\Source\portable\MemMang`.

Some FreeRTOS components call user-implemented functions, such as hook functions and callback functions. Next, we'll create a C file to hold those functions:

- In the project, create this directory for storing the file: `Config/FreeRTOS/Src`.
- Create an empty C file, `freertos.c`, in the directory: `Config/FreeRTOS/Src/freertos.c`.

The file's name and where it is stored are up to the user. This is just what we chose.

For our FreeRTOS configuration, two callback functions must be provided. We'll put them in `freertos.c`. Our `FreeRTOSConfig.h` file specifies that static allocation is supported:

```
#define  configSUPPORT_STATIC_ALLOCATION    1
```

Static allocation requires two callback functions. They can be found at: `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/02-Customization#configsupport_static_allocation`

- Copy those two callback functions into `freertos.c`. The functions are:
    - `vApplicationGetIdleTaskMemory()`
    - `vApplicationGetTimerTaskMemory()`

- Two `#include` statements must also be added to the file:

```
#include "FreeRTOS.h"
#include "task.h"
```

Configure the build-system for FreeRTOS

The project's build system must be configured for the include-files and source-files that we just added.

Relative file-paths should be specified, for the project to be portable, and so it can be copied.

The build configuration can be opened via Project Explorer. Right-click on the project and select **Properties**.

In these instructions, we'll be setting up the `Debug` build-configuration. There is also a `Release` build-configuration, but it is not set up here.

First, we'll add the include paths for the new include-files, as shown below:

- In the left-most pane, select: **C/C++ Build** and then **Settings**
- Select the tab **Tool Settings**
- In the tab's left-most pane, select: **MCU GCC Compiler** and then **Include paths**
- In the **Configuration** drop-down box, select **Debug**, as shown in *Figure C.13*:

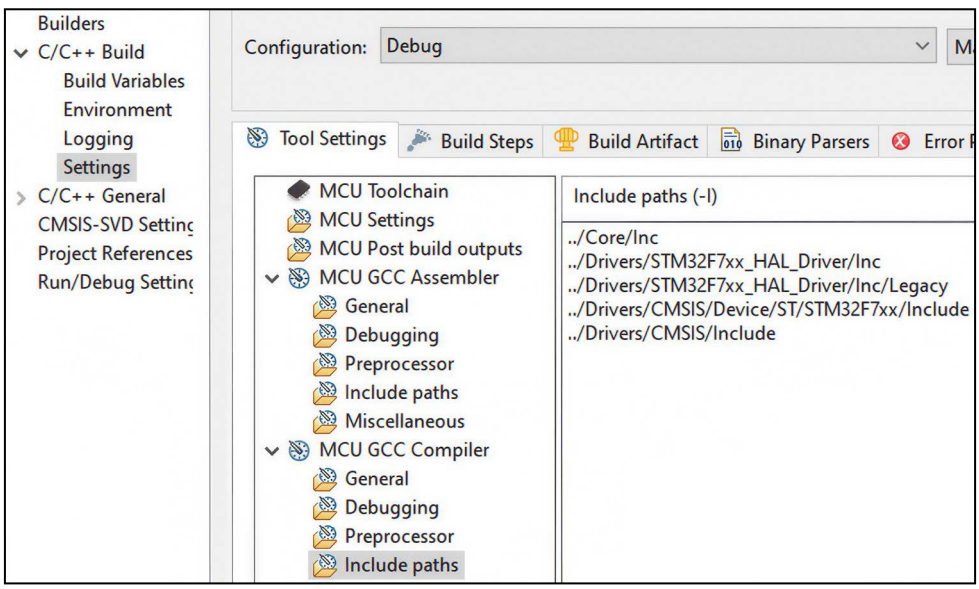


Figure C.13: Include-paths in the build configuration

- In the **Include paths** pane, to add an include-path:
  - Click on the **Add** icon, **+**.
  - Add the three include-paths shown below. The **Include paths** pane should look like *Figure C.14*:

```
../Middlewares/Third_Party/FreeRTOS/Source/include
../Middlewares/Third_Party/FreeRTOS/Source/portable/GCC/ARM_
CM7/r0p1
../Config/FreeRTOS/Inc
```

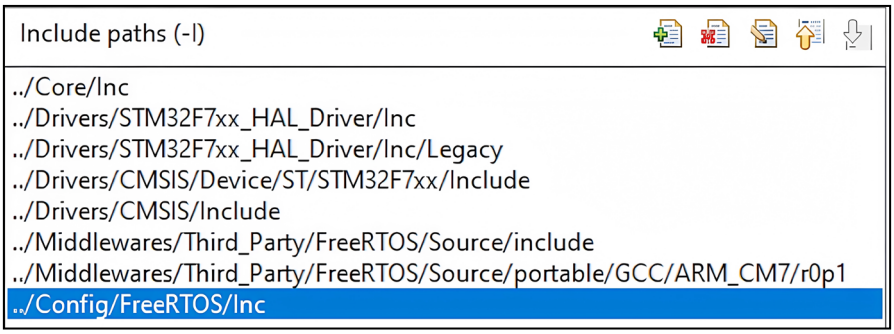


Figure C.14: Showing the three added include-paths

In the earlier section *Uninstall FreeRTOS*, the directory `CMSIS_RTOS_V2` was deleted, if present. Its include-path will also need to be deleted, if present:

- Under **Tool Settings**, select **MCU GCC Assembler** and then **Include paths**.
- Delete this include path, if present:

```
../Middlewares/Third_Party/FreeRTOS/Source/CMSIS_RTOS_V2
```

Next, we'll add source folders to the build-path.

- In the left-most pane, select **C/C++ General**, and then **Paths and Symbols**.
- In the **Configuration** drop-down box, select **Debug**.
- Select the tab **Source Location** (shown in *Figure C.15*).

- Add the following two paths, which are also shown in *Figure C.15*:
    - `/FreeRTOS_project/Config/FreeRTOS/Src`
    - `/FreeRTOS_project/Middlewares`

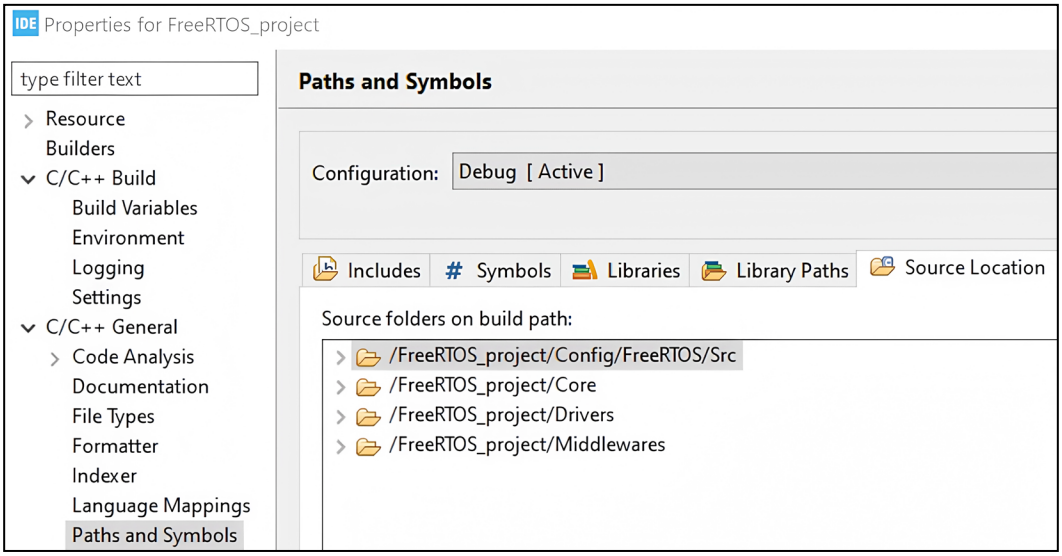


Figure C.15: Adding source folders to the build-path

Having installed FreeRTOS, we can now test the IDE-project.

Writing and running a FreeRTOS program

To test our FreeRTOS install, we'll create and run a simple FreeRTOS app.

Writing a FreeRTOS app

An easy way to write a FreeRTOS app is to add to the `main.c` that is generated by the IDE. In our project, it's at `Core\Src\main.c`. Most of the book's example-programs are written by adding to a similar `main.c`. There, in the function `main()`, FreeRTOS APIs are called to create tasks and start the scheduler. Also, the tasks are defined in `main.c`.

Since our project's `main.c` is generated by the IDE, it has `USER CODE` sections for adding code. If the project's hardware configuration is ever changed, then `main.c` would be re-generated. However, whatever is in the `USER CODE` sections will not be modified.

An example FreeRTOS-program is in the book's code-repository on GitHub, at the path below (the downloaded code-repository is described in chapter 3):

```
packtBookRTOS2E\templates\Appendix-C--FreeRTOS-example--main.c
```

That example-program is a `main.c` that was generated by the IDE, using the installation process that was just presented. Code has been added to its `USER CODE` sections. Code comments explain how the user-code works. Two tasks are created and run by the scheduler. Each task blinks an LED.

We can use the example-program to create our simple FreeRTOS app. Just copy the example-program's `USER CODE` sections into our project's `main.c`, at the corresponding locations. Within the `USER CODE` sections, copy the code and code-comments. We'll use this updated `main.c` to test our FreeRTOS install. It will also be used later, to test our SystemView install. Among the code-comments is code for using SystemView, but that code is commented-out. We'll uncomment that code later, after we install SystemView.

Building and running the FreeRTOS app

After creating our test app in `main.c`, the next steps are to build it and run it.

To build the app:

- Select the Debug build-configuration:
    - In Project Explorer, right-click on the project
    - Select: **Build configurations | Set active | Debug**
- Build the project:
    - Right-click on the project, and select **Build project**

Run the program using Ozone

- Create the `*.jdebug` file (described in chapter 3)
- In using the template `*.jdebug` file, ensure the line for FreeRTOS is uncommented.
- Download and run the program

When the program runs, the dev-board's blue LED should be on, and the red and green LEDs should flash.

With that, we've successfully installed FreeRTOS and tested the installation. There's just one remaining part in the installation process.

Installing SystemView

The following sections describe how to install SystemView, for use with FreeRTOS:

- Install the SystemView source-code
- Patch the FreeRTOS source-code
- Configure the build-system for SystemView
- Use SystemView

Before installing SystemView, it's recommended that a backup be made of `FreeRTOS_project`.

Install the SystemView source-code

The SystemView program that runs on the MCU is distributed as source code. And, it's referred to as the **SystemView Target Sources**. There is a SystemView app that runs on the development computer, and it is assumed to be version 3.60d.

For these installation instructions, we'll be using Target Sources version V3.60d. Download `Target Sources` here: `https://www.segger.com/downloads/systemview/`.

- Unzip the downloaded file. In the expanded folder, the top-level folder is `SystemView_Src_V360d`.
- Copy that top-level folder to `FreeRTOS_project`, under the directory `Middlewares\Third_Party`.
- Rename the folder to `SystemView`.

The `Middlewares` directory should now look like this:

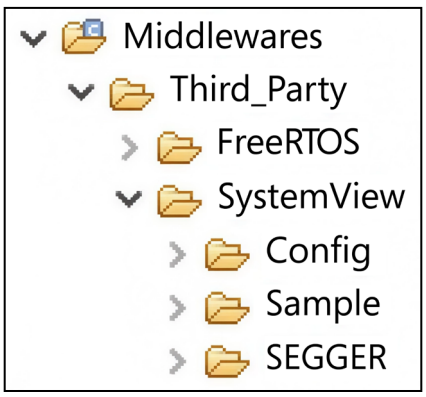


Figure C.16: The SystemView Target Sources, copied to the IDE-project

The folder `SystemView\Sample` has sub-directories for the supported OSes, including FreeRTOS, as shown in *Figure C.17*. Each OS directory has initialization and configuration code for the OS specified in the directory name, e.g., `FreeRTOSV11`. We need to choose the most appropriate OS directory for our install, and we'll delete the others.

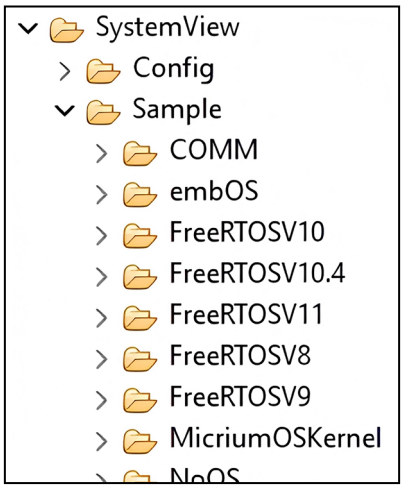


Figure C.17: The directory SystemView\Sample

There are several directories for FreeRTOS, and the directory name specifies the applicable kernel version. We'll use the FreeRTOS directory whose kernel version is the highest, but not more than the installed FreeRTOS kernel. Our installed FreeRTOS kernel is version 11.1.0, so we'll use the directory `FreeRTOSV11`.

In the directory `SystemView\Sample`, delete all the sub-directories except `SystemView\Sample\FreeRTOSV11`. The directory `Sample` should now look like this:

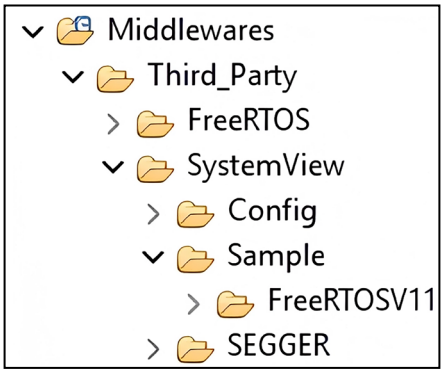


Figure C.18: The directory SystemView\Sample, containing only FreeRTOSV11

Add the following `#include` to the end of the file `Config\FreeRTOS\Inc\FreeRTOSConfig.h`, inside the `USER CODE` section, as shown below:

```
/* USER CODE BEGIN Defines */
/* Section where parameter definitions can be added. . . */
#include "SEGGER_SYSVIEW_FreeRTOS.h"
/* USER CODE END Defines */
#endif /* FREERTOS_CONFIG_H */
```

Patch the FreeRTOS source-code

Since we're using the sample `FreeRTOSV11`, the FreeRTOS source-code does not need to be patched. Currently, this is not documented in the SystemView user-guide. However, it is documented in Segger's "Knowledge Base" for SystemView: `https://kb.segger.com/FreeRTOS_with_SystemView`.

When using a FreeRTOS prior to version 11, the appropriate SystemView sample must be used. And, the FreeRTOS source-code will need to be patched, using a patch file provided in the sample. More information is in the SystemView user-guide and "Knowledge Base".

Again, though, for the present instructions, the FreeRTOS source-code does not need to be patched.

Also, the SystemView "Knowledge Base" mentions adding defines to `FreeRTOSConfig.h` for `INCLUDE_xTaskGetIdleTaskHandle` and `INCLUDE_pxTaskGetStackStart`. However, the specified defines do not need to be added for FreeRTOS version 10 and later.

Configure the build-system for SystemView

The project's build-system needs to be configured for the added SystemView include-files. For the added SystemView source-files, the build-system is already configured, since they are under the `Middlewares` directory.

An earlier section showed how the build-system is configured for the FreeRTOS include-files. (That section is *Configure the build-system for FreeRTOS*.) The same configuration process can be used here.

In the build-system, add these include paths to the **MCU/MPU GCC Compiler**:

- `../Middlewares/Third_Party/SystemView/Config`
- `../Middlewares/Third_Party/SystemView/Sample/FreeRTOSV11`
- `../Middlewares/Third_Party/SystemView/SEGGER`

Add this include path to the **MCU/MPU GCC Assembler**:

- `../Middlewares/Third_Party/SystemView/Config`

Use SystemView

Next, we'll test the SystemView installation by using it with our running system.

In `main.c`, code was added earlier that calls FreeRTOS APIs. The code is in the `USER CODE` sections. The added code includes comments with calls to SystemView APIs. Uncomment that code, to test the SystemView install. The commented-code is identified by the line-comment:

```
// SystemView example-code:
```

`Task1` is the highest-priority task. When `Task1` starts, it will spin until the SystemView app is started, in Record mode. The tasks continually toggle LEDs, and the tasks write messages by calling `SEGGER_SYSVIEW_PrintfHost`.

The SystemView "Knowledge Base" has instructions for installing SystemView. They state, "add `traceSTART();` to main." That API appears to be deprecated. In the SystemView Target Sources, `traceSTART` is redefined to be `SEGGER_SYSVIEW_Conf`. In our `main.c`, there is a call to `SEGGER_SYSVIEW_Conf`, within the commented-code.

The project can be built and run via Ozone. Once running, the SystemView app can be started, in Record mode.

In SystemView, the **Events List** will be filled mostly with uninteresting `SysTick` events. Unwanted events can be hidden using **View | Event Filter | Hide This Event**. For the selected event, all events of that type will be hidden. *Figure C.19* shows three types of `SysTick` events that can be hidden.

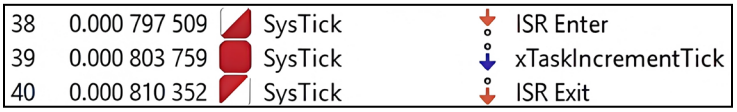


Figure C.19: SysTick events to be hidden

Installation summary

We now have a working IDE-project with FreeRTOS and SystemView installed. (An instance of it is in the book's code-repository, and it is named `FreeRTOS_project`.) In the IDE-project, FreeRTOS is configured for general-purpose use. It may be necessary to make changes to `FreeRTOSConfig.h` for your particular system. For example, for many FreeRTOS APIs, they have to be enabled in `FreeRTOSConfig.h` in order to use them. The `FreeRTOSConfig.h` that we installed enables typical APIs, but not all. In the FreeRTOS reference manual, each API's documentation specifies how to enable it. Information on enabling other FreeRTOS features can be found on the `FreeRTOS.org` web-site. Relevant web pages are listed in *Appendix B, Reference Information*.



www.packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

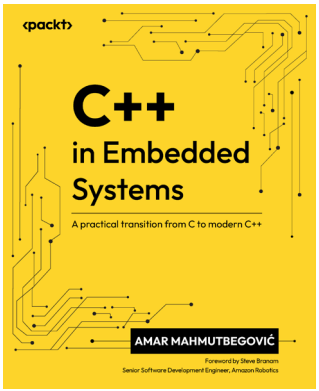
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

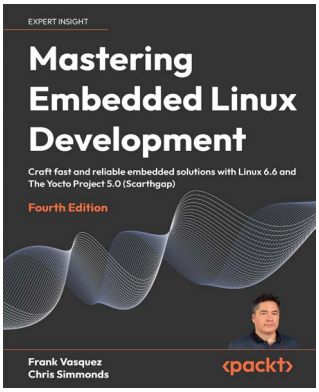


C++ in Embedded Systems

Amar Mahmutbegović

ISBN: 978-1-83588-114-9

- Debunk myths and misconceptions about using C++ in embedded systems
- Set up build automation tailored for C++ in constrained environments
- Leverage strong typing to improve type safety
- Apply modern C++ techniques, such as Resource Acquisition Is Initialization (RAII)
- Use Domain Specific Language (DSL) with a practical example using Boost SML
- Implement software development best practices, including the SOLID principle, in embedded development



Mastering Embedded Linux Development, Fourth Edition

Frank Vasquez, Chris Simmonds

ISBN: 978-1-80323-259-1

- Cross-compile embedded Linux images with Buildroot and Yocto
- Enable Wi-Fi and Bluetooth connectivity with a Yocto board support package
- Update IoT devices securely in the field with Mender or balena
- Prototype peripheral additions by connecting add-on boards, reading schematics, and coding test programs
- Deploy containerized software applications on edge devices with Docker
- Debug devices remotely using GDB and measure the performance of systems using tools like perf and ply

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Hands-On RTOS with Microcontrollers, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

analog-to-digital converters (ADCs) 5, 26, 69

application-specific integrated component (ASIC) 8

ARM Cortex-M processors 71

ARM FreeRTOS firmware stack 17

assertions
- assert_param() 462
- configAssert() 462, 463
- using, for debugging FreeRTOS 462

asymmetric multi-processing 441

B

bare-metal drivers 341

bare-metal firmware 9

binary semaphores 191, 192
- synchronization via 192-196

buffer-based driver 293
- performance analysis 298-300
- uartPrintOutTask 294
- USART2_IRQHandler 295-297
- with DMA 313-315

C

CDC_Transmit_FS() 349

central processing unit (CPU) 25

C function 151

command-execution task 392-395

command queue
- architecture 390, 391
- code, using 400, 401
- components 387
- creating 386
- deciding 389, 390
- LEDs' brightness functions 387-389

command queue, architecture
- command-execution task 392-395
- frame decoding 395-398
- USB VCP driver 399

complex programmable logic devices (CPLDs) 8

components, dev-board 24
- additional components 27
- central processing unit (CPU) 26, 27
- microcontroller unit (MCU) 24-26
- ST-LINK debug probe 28-31

composite data-type 244

computer numerical control (CNC) 13

concurrent execution 126

configAssert() 462
data collection 464-466
hung system, debugging with 463, 464
Ozone data breakpoints 466-470

context switch 127

controller area network (CAN) 18, 456

Cortex-M7 26, 27
floating point unit (FPU) 27
hardware debug features 27
memory protection unit (MPU) 27
nested vector interrupt controller (NVIC) 27
SysTick timer 27

counting semaphores 188-190, 205, 206

D

data passing, through queues by reference 250
passing by reference 252-255
passing by value 252
scenario 251

data passing, through queues by value 240
composite data-type by value 244-247
example 250
one byte by value 241-244
queues , affecting execution 247-249

debugger 28

debugging tools 49
SEGGER J-Link, using 50
SEGGER Ozone, using 51
SEGGER SystemView, using 58
troubleshooting 472

debug probe 29

delay 67

design and testing considerations, of super-loop example-program 94
debugging 95
timing resilience 94

dev-board 21, 22
architecture 23
components 24

direct memory access (DMA) 180, 303-307
buffer-based driver with 313-315

DMA-based driver
creating 304
performance analysis 315-319

DMA controller 305
peripheral, configuring 307-311

DMA receive
beginning 312, 313

double-buffered DMA 323-325

driver model
selecting 334
selecting, considerations 335-339
selecting, rules of thumb 339, 340

dynamic allocation
eliminating 422

dynamic allocation, of FreeRTOS primitives 418
queue, creating 420
task, creating 419

dynamic allocation of memory 176

dynamic memory-allocation 412

E

electromagnetic interference (EMI) 451

enhanced DMA-based driver
configuration and initialization 325-327
creating 319
data, receiving from stream-buffer 327, 328

double-buffered DMA 323-325
implementing 325
performance, analyzing 330-333
stream-buffer, improving 334
stream-buffer, populating 328-330
stream buffers 319

Ethernet 456

event-detection delay 67

example-programs directory 42

example-programs, STM32CubeIDE
downloading 42, 43
importing into 43-45
installing 41
organizing 41

Executable and Linkable Format (ELF) 52

F

field-programmable gate arrays (FPGAs) 8

file-types
*.bin 52
*.elf 52
*.hex 52
*.svd 52

firmware 26

floating point unit (FPU) 27

frame protocol-decoding 395-398

FreeRTOS 16, 17
app, building and running 526
app, writing 525, 526
build-system, configuring for 523-525
configuring 146-148
generating 518, 519
hardware, configuring to run 512-515
installing 146, 512
source-code, installing manually 521-523
uninstalling 520-521

FreeRTOS APIs
calling, from ISRs 281

FreeRTOS App 346

FreeRTOS application
additional Ozone features 162, 163
code, building 161
code, running 161
SystemView, using to analyze task execution 164-166

FreeRTOSConfig.h
configuring 515-518
copying 519
creating, with IDE 515

FreeRTOS debugging-techniques 462
assertions, using 462
memory-use, monitoring 471
stack-overflow, detecting 472
tools, for analyzing tasks and ISRs 471

FreeRTOS, example-program
IDE-project 133
running 138
scheduler 134
SystemView, starting 136, 137
SystemView, using 138-145
Task1, working 135, 136
Task2 and Task3, working 137
tasks, creating 134
working 133

FreeRTOS heap implementations 423-425
comparing 426
selecting 426-428

FreeRTOS primitives
dynamic allocation 418
static allocation 420

FreeRTOS project
IDE-project, creating 512
installation process 511, 512

installing 509
overview 509, 510
FreeRTOS scheduler 120, 167
architecture 174-176
creating 150
Idle task 169-172
running 172, 173
selecting, as RTOS 131, 132
starting 158
super-loop functions, changing to tasks 123
super-loops limitations 120, 121
tasks, versus super-loop functions 122
working, with example-program 132
FreeRTOS tasks
creating 150-156
deleting 158
deleting, from another task 160
hardware, initializing 151
return value, checking 156, 157
task functions, defining 151-154
vTaskDelete() 159
FreeRTOS-tasks
interrupt features, configuring 281-285
versus ISRs 278-280
FreeRTOS task states 167
blocked state 168, 169
ready state 168
running state 168
suspended state 169

G

general-purpose operating system (GPOS) 440
graphics processing units (GPUs) 438

H

handler mode 174
Hardware Abstraction Layer (HAL) 26, 87, 97
hardware-based real-time systems 8
hardware documentation 32, 33
hardware timers
downsides 224
heap
space 432, 433
heap fragmentation 417
heap memory 415, 416
heterogeneous multi-core systems 440-443
inter-core communication 443
legacy application extension 443-447
high-demand hard real-time systems 447-449
high-reliability systems 453
homogeneous multi-core systems 449

I

IDE-project
creating 512
creating, for using in learning FreeRTOS 505, 506
in-circuit emulator (ICE) 31
inner loop 65
integrated circuit (IC) 24
integrated development environment (IDE) 27, 39
editor, using 47
example-programs 39
project, building 48, 49
Project Explorer, using 46, 47
STM32CubeIDE 40

STM32CubeIDE, installing 40
used, for creating FreeRTOSConfig.h 515

inter-integrated communication bus (I2C) 456

inter-processor communication 453
communication standards 456
considerations 453
development team/project constraints 455
physical requirements 454
timing considerations 454

inter-processor communication, standards 456
controller area network (CAN) 456
Ethernet 456
inter-integrated communication bus 456
local interconnect network (LIN) 457
Modbus 457
serial peripheral interface (SPI) 457
USB as inter-processor communication bus 457

interrupt features
configuring, in FreeRTOS 281-285

interrupts 68
super-loops 71, 72

Interrupt Service Routines (ISRs) 8, 27, 67, 68, 263
FreeRTOS APIs, calling from 281
linking 289
linking, tips 290, 291
versus FreeRTOS-tasks 278-280

interrupt status register (ISR) 272

intertask communication
queues, using 238-240

iPWM interface 404

ISR-based drivers
buffer-based driver 293-300
creating 285
queue-based driver 285-293

ISR-based UART-driver 271

iteration execution time 65

iteration execution-time 93

iteration frequency 65

iteration period 65, 93

J

jitter 67-69

J-Link debug probe 32

J-Link USB-cable 344

K

KISS principle 7

L

learning FreeRTOS
IDE-projects, creating to use in 505, 506

LED command-queue
iPWM interface 404
LED command-queue 404

local interconnect network (LIN) 457

loop iteration 65

loose coupling 341

M

main-stack pointer (MSP) 414

malloc
replacing 428, 429

MCU-controlled soldering iron 5

MCU FreeRTOS system
FreeRTOS heap 415
system heap 415
memory allocation 412
memory management unit (MMU) 10, 441
memory pools 424
memory protection unit (MPU) 27, 177, 433
using 434
microcontroller unit (MCU) 23-26, 437
families 33, 34
model names 34, 35
middleware 15
MISRA-C 412
Modbus 457
multi-core systems 438
heterogeneous multi-core system 440-443
high-demand hard real-time systems 447-449
homogeneous multi-core systems 449
scenarios 458
multi-processor systems 439
design reuse 453
distributed systems 450, 451
high-reliability systems 453
parallel development 452
scenarios 459
mutex-acquisition timeouts
avoiding 218-220
mutexes 12
mutual exclusion (mutex) 213-215
example-program, for fixing priority-inversion 215-217
mutex-acquisition timeouts, avoiding 218-220
used, for avoiding race conditions 220

N

Nested Interrupt Vector Controller (NVIC) 281
nested vector interrupt controller (NVIC) 27, 269
Nested Vector Interrupt Controller (NVIC) 68
newlib-nano 428
NUCLEO
reference link 35
NUCLEO-144 35
NUCLEO-F207ZG 21
Nucleo-F767ZI 21
NUCLEO-F767ZI
reference link 35
NVIC_EnableIRQ 287
NVIC_SetPriority 287

O

one-shot timers 226, 227
oscilloscope
connecting, to LED 113-115
running 115, 116
using, to measure timing 112
Ozone 488
problems, troubleshooting 472-476

P

polled UART-driver
advantages 276
creating 271
disadvantages 276
example-program 271-273
example-program, running 273, 274
performance, analyzing 275, 276
usage 277

polling 67
synchronization by 197-200
practical real-time systems
FreeRTOS 16, 17
range of RTOSes 15, 16
RTOS 16
preemptive-based scheduling 128-130
primitive atomic-instructions 224
printed circuit board assembly (PCBA) 439
priority inheritance 214
priority-inversion 206-208
example-program with 208-213
fixing with mutex,
example-program 215-217
process-stack pointer (PSP) 414
programmable logic controllers (PLCs) 24
programmable logic devices (PLDs) 8
project 487
building 488
source-code files 487
project-directory 41
Project Explorer 46
proportional-integral-derivative (PID) 7
Pulse Width Modulation (PWM) 387

Q

queue-based driver 285
ISRs, linking 289-291
performance analysis 291-293
startReceiveInt function 286, 287
startUart4Traffic 291
uartPrintOutTask 286
USART2_IRQHandler 287-289
queue definition
reusing, for new target 402, 403
queues
as interfaces 405, 406

R

race conditions 220
avoiding, with mutexes and
semaphores 220
example 220-224
random number generator (RNG) 85
real-time systems 4
bare-metal firmware 9
carefully crafted OS software 11
failure scenarios, addressing 7
hardware-only 8
practical real-time systems 14
range of timing requirements 5-7
real-time requirements 4
RTOS-based firmware 9
RTOS-based software 10
timing failure 4
receive data register (RDR) 272
receive not empty (RXNE) 272
repeat timers 227-230
Reset and Clock Control (RCC) 269
round-robin scheduling 126, 127
RTOS-based firmware 9, 10
RTOS-based software 10
RTOSes
defining 11-13
firm real-time systems 14
hard real-time system 13
selecting 18, 19
soft real-time systems 14

RTOS queues 234
empty queue 237
for intertask communication 238-240
full queue 236
simple queue 234, 235

S

scheduler 122
RTOS scheduler 130
also see FreeRTOS scheduler
SEGGER J-Link
ST-Link, converting to J-Link 51
using 50
utilities, download and installation 50
SEGGER Ozone
downloading 53
file-types, using 52, 53
installing 53
project-file (.jdebug), configuring 54
source-code debugger, using 57, 58
used, for downloading program 55, 56
using 51
SEGGER SystemView
installing 58, 59
running 60, 61
using 58
semaphores 188
binary semaphores 191, 192
counting semaphores 188-190, 205, 206
synchronization, by polling 197-200
synchronization, via binary semaphores 192-196
time-bound binary semaphores 201-204
used, for avoiding race conditions 220
using 192
Serial-Console App 347
serial peripheral interface (SPI) 457
shared LEDs 208
shared-resource 208
software timers 224, 225
considerations 230
example-program, with one-shot timers 226, 227
example-program, with repeat timers 227-230
guidelines 230
guidelines, example use cases 230
limitations 230, 231
stack 17
space 429-431
stack memory 413, 414
standardized API 17
startReceiveInt function 286, 287
static allocation, of FreeRTOS primitives
queue, creating 421, 422
task, creating 420
static allocation of memory 176
static memory 413
ST-LINK 50
ST-LINK debug probe 28-31
ST-LINK/V2-1 29-31
STM32
reference link 34
STM32CubeIDE 29, 487
installing 41
project, copying and renaming 506-509
STM32Cube USB Device-Library 346
STM32F 35
STM32F7
reference link 35
STM32F7x7
reference link 35

STM32F76x 35

STM32F767 35

STM32F767ZI 33
connectivity peripherals 26
direct memory access (DMA) 26
GPIO 26
reference link 35
timers 26

STM32F767ZI MCU 25, 26

STM32F767ZI model 34

ST Nucleo-F767ZI 24

stream batching buffer 322

stream buffers 303, 319
creating 320
data, receiving from 321, 322
data, sending to 320
improving 334

super-loop example-program 84-93
build-configuration settings, adding 110-112
creating 102
debugger software, configuring 108
debugger software, installing 108
design and testing considerations 94
developing 101
firmware, building 112
firmware, running 112
hardware, configuring 104-107
IDE project, creating 102
initialization 87-90
oscilloscope, using to measure timing 112
processor utilization 93
requisites 84, 85
running 96, 97
SystemView target source-code, installing 109, 110
user-code, creating 107, 108
working 85, 86

super-loop functions, modifying into RTOS tasks
conceptual task-programming model 123-126
preemptive-based scheduling 128-130
round-robin scheduling 126, 127

super-loops 64, 65
functions 65
in real-time systems 66, 67
interrupts 71, 72
timer, adding 72, 73
timing-related performance analysis 80, 81

symmetric multi-processing 449

system APIs and hardware features 97
__disable_irq 99-101
HAL_DBGMCU_EnableDBGSleepMode 97-99

system heap 415

system tick 127

SystemView 489, 490
build-system, configuring 529
FreeRTOS source-code, patching 529
installing 509, 527
problems, troubleshooting 479-483
source-code, installing 527, 528
using 530

SysTick timer 27, 87

T

Task Control Block (TCB) 158, 256, 415

task implementation and troubleshooting 176
task memory allocation 176
task startup problems troubleshooting 181
task states optimization 179

task memory allocation 176
creation roundup 178, 179
dynamically allocated tasks 176
memory-protected tasks, creating 177, 178
statically allocated tasks 177
task notifications 255
indexed task-notifications 260
notification state 256
notification value 256
other actions 258
simple data, passing 256, 257
versus queues 258, 259
task startup problems troubleshooting 181
scheduler returns unexpectedly 183
task creation failed 181, 182
task starvation 129
task states optimization 179
increase performance, optimizing 180
minimize power consumption, optimizing 180
reduce CPU time, optimizing 179, 180
Tera Term 346-348
third-party libraries and drivers
using 340, 341
thread mode 174
threads 12
time-bound binary semaphores 201-204
timed super-loop
putting, to sleep 74, 75
using 76-80
timeout 13
timer
adding, to super-loop 72, 73
timing assurance 94
transfer-halted time 318

U

universal asynchronous receiver/transmitter (UARTs) 263-267
setting up 267-271
universal synchronous/asynchronous receiver/transmitter (USARTs) 265
unmanned aerial vehicle (UAV) 7
USB
messages, sending 344-348
USB API
code 349, 350
code, analyzing 352, 353
code, improving 353, 354
code, running 351, 352
messages, sending 349
USB Device-Controller 346
USB Host-Controller 347
USB VCP driver 399
User USB-connector 345

V

VCP driver 356-360, 365
analyzing 361
code, running 360, 378, 379
creating 354, 355
enhancing 362-364
performance, analyzing 380, 381
performance-data 376
vcpInit() function 366
vcpSend() function 369
data, sending to stream-buffer 375
initialization 370
initial processing 369
maximum wait-time, implementing 371, 372

parameters 370
return value 370
semaphore, using to wait for
stream-buffer space 373-375

vcpTransmitComplete() function 368

vcpTransmitTask() function 367

Virtual COM Port (VCP) 347, 400

vTaskDelete() 159

W

Windows USB-drivers 347

workspace 487

workspace-directory 43

X

xQueueCreate() function 241

xTimerCreate 226