Zephyr RTOS Embedded C Programming

Using Embedded RTOS POSIX API

Andrew Eliasz

Apress®

Zephyr RTOS Embedded C Programming

Using Embedded RTOS POSIX API

Andrew Eliasz

Apress®

Zephyr RTOS Embedded C Programming: Using Embedded RTOS POSIX API

Andrew Eliasz
First Technology Transfer
Croydon, Surrey, UK

ISBN-13 (pbk): 979-8-8688-0106-8 ISBN-13 (electronic): 979-8-8688-0107-5
https://doi.org/10.1007/979-8-8688-0107-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Susan McDermott
Development Editor: Laura Berendson
Project Manager: Jessica Vakili

Cover designed by eStudioCalamar

Cover image by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on the Github repository: https://github.com/Apress/Zephyr-RTOS-Embedded-C-Programming. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

Table of Contents

About the Author

Andrew Eliasz is the Founder and Head at Croydon Tutorial College as well as the Director of First Technology Transfer Ltd. First Technology Transfer runs advanced training courses and consults on advanced projects in IT, real-time, and embedded systems. Most courses are tailored to customers' needs. Croydon Tutorial College evolved from Carshalton Tutorial College, which was established to provide classes, distance-level teaching, workshops, and personal tuition in computer science, maths, and science subjects at GCSE, A Level, BTEC, undergraduate, and master's levels. It has now changed its name and location to Croydon Tutorial College at Weatherill House, Croydon. In addition to teaching and tutoring, they also provide mentoring and help for students having difficulties with assignments and projects (e.g., by suggesting how to add to a project to obtain a better grade as well as reviewing project content and writing styles).

About the Technical Reviewer

Jacob Beningo is an embedded software consultant with over 15 years of experience in microcontroller-based real-time embedded systems. After spending over ten years designing embedded systems for automotive, defense, and space industries, Jacob founded Beningo Embedded Group in 2009. He has worked with clients in more than a dozen countries to dramatically transform their businesses by improving product quality, cost, and time to market. Jacob has published more than 500 articles on embedded software development techniques and is a sought-after speaker and technical trainer who holds three degrees that include a Master of Engineering from the University of Michigan. He is an avid writer, trainer, consultant, and entrepreneur who transforms the complex into simple and understandable concepts that accelerate technological innovation. Jacob has demonstrated his leadership in the embedded systems industry by consulting and training at companies such as General Motors, Intel, Infineon, and Renesas along with successfully completing over 50 projects. He holds bachelor's degrees in Electronics Engineering, Physics, and Mathematics from Central Michigan University and a master's degree in Space Systems Engineering from the University of Michigan.

In his spare time, Jacob enjoys spending time with his family, reading, writing, and playing hockey and golf. In clear skies, he can often be found outside with his telescope, sipping a fine scotch while imaging the sky.

CHAPTER 1

An Introduction

What This Book Is "All About"

This book is a foundational guidebook introducing programming embedded and IoT/IIoT (Internet of Things/Industrial Internet of Things) applications in C using the Zephyr RTOS framework. It is for engineers and programmers planning to embark on a project involving the use of Zephyr RTOS, or evaluating the potential advantages of using Zephyr RTOS in an upcoming project.

You, the reader, probably have a digital electronics and embedded systems programming background building specialized embedded systems applications in C and assembler. Maybe the requirements of upcoming applications are such that a classical bare metal programming approach may not be the best way to go. Maybe you have inherited some poorly documented complex multitasking code and the developers or consultants involved in developing this code have left the project and your company is considering migrating the code to use a real-time multitasking operating system.

The aims of this book are to show you what Zephyr is capable of and to introduce you to the basic RTOS programming skills required before embarking on a real-world real-time RTOS-based project. The book can also be thought of as a guide to the rich and complex framework that makes up Zephyr RTOS and to the examples that are part of the Zephyr code repository.

A. Eliasz, *Zephyr RTOS Embedded C Programming*,
https://doi.org/10.1007/979-8-8688-0107-5_1

Alternatively, you may have embedded Linux programming experience and have to develop applications on processors that, though powerful, are too small to run a full Linux system. Here, again, one of the things that makes Zephyr special is that it has embraced and adapted many of the concepts and technologies that make Linux so special, things such as support for the POSIX API and the use of Linux technologies such as Kconfig and devicetree.

What Is an RTOS and When and Why "Do You Need One"?

Modern microcontrollers come in a wide variety of sizes and complexity ranging from 8-bit microcontrollers with less than 10 kilobytes (10K) Flash and less than 2 kilobytes (2K) RAM through to multiprocessor 64-bit microcontrollers interfacing with gigabytes of memory. There are SoC (System on Chip) processor architectures at the lower end of the embedded computing spectrum and SoM (System on a Module) boards at the upper end.

For tiny systems performing a single specialized task, or a small number of fixed tasks, such as a motor controller in a toothbrush or power drill controlling a motor, the code can be implemented as a bare metal application. The complexity of modern connected applications means that they are not best suited to being implemented as bare metal applications. Modern microcontroller vendors often provide IDEs that provide a graphical interface for configuring peripherals and "pulling in" driver code into the project, thus allowing developers to focus on the application they are trying to build. Examples include Microchip's Harmony tool and STMicroelectronics STM32CubeIDE. Embedded systems applications can also be developed using an IDE such as Microsoft's VS Code with suitable plug-ins.

An operating system can be thought of as software that provides services that can be used for developing applications where multiple pieces of work (tasks) have to be worked on concurrently. At the center of an operating system is the scheduler, whose job is to decide which task is to run next. In a cooperative multitasking operating system, a task runs till it decides to suspend what it is doing and transfer control to the scheduler, which will determine which task to run next. In a preemptive multitasking operating system, a task can be preempted by the operating system at any point. Preemption may occur because a higher priority task is ready to run, or because the running task needs to access a resource that is currently not available because it is being used by another task. The concept of Real Time refers to how long it takes the system to respond to some event, such as a button press, or arrival of data at a communications peripheral, or completion of an ADC (Analog to Digital) conversion. A distinction is often made between hard and soft real-time systems. In a hard real-time system, it is an error if the time taken for a response exceeds some specified duration. In a soft real-time system, the response time is interpreted in a statistical sense in which most of the time the required time-to-completion limits are met, but, occasionally, they are not.

Classical bare metal multitasking, typically, involves a combination of a "superloop" that handles non-time-critical work, with time-critical work being done in interrupt handlers. The classical Arduino IDE also follows this pattern.

In the modern world of networked devices (both wired and wireless networking) running relatively complex network protocol stacks and doing so in a secure manner, the standard bare metal approach runs into difficulties. A networked device may have several interfaces, for example, wired or wireless Ethernet, USB, and serial communications such as CAN bus, RS232, or RS485. The code involved is quite complex, and having to handle the low-level details together with the other tasks being performed by the device, such as, for example, taking sensor readings on a periodic basis, adds further complexity. A networked device may have to interact

with a number of other devices, and the communication traffic patterns may be unpredictable. Worse still traffic may be bursty, and the system will need to protect itself against overloading by heavy bursts of traffic.

Packet-oriented communications protocols such as TCP/IP are multilayered, and a packet will contain multiple headers corresponding to the various layers and the functionality they provide. It is not uncommon for protocols to support multiplexing. For example, the TCP/IP stack handles both UDP and TCP traffic as well as ICMP traffic, and in the case of UDP and TCP, there may be traffic associated with different processes running on the device each identified by a particular identifier (port number).

From the design and implementation point of view, a multitasking approach allows the various tasks to be worked on separately and then combined together, courtesy of the scheduling and intertask communication and synchronization mechanisms such as semaphores and message queues provided by the RTOS.

The key motivation underlying the use of an RTOS to build embedded applications is that it provides a framework and its associated abstractions, APIs that support developing code that can handle the time, priorities, and preemptibility of the tasks that constitute that application so that task deadlines can be met and the system exhibits deterministic behavior. From a developer's point of view, an RTOS can be thought of as providing services, not only scheduling, synchronization, and intertask communication services but also, if required, file systems services, communications services, and security services.

What Is an RTOS?

The OS in RTOS stands for Operating System. An operating system can be thought of as a collection of modules (libraries) that provide task scheduling and control services, where a task is code that carries out a

particular piece of the overall application's work. A modern advanced operating system will also provide device drivers for widely used devices and peripherals, communications protocol stacks and application layer modules on which actual applications can be layered, security and memory protection or memory management services, and much more besides. The RT in RTOS stands for Real Time.

Real Time here refers to predictable and reproducible behavior. This behavior may be predictable in a statistical sense, for example, where the response times to some event will follow a statistical distribution with a certain mean and variance. This is "soft" real time. For certain applications, there may be a requirement that the response time is always less than some specified value. Such applications are referred to as "hard" real-time applications. It is also possible to have systems that involve both "hard" and "soft" real-time aspects.

Using Open Source RTOS in Systems Requiring Functional Safety

In the case of applications where a high degree of functional safety is involved, the question also arises as to whether open source software can be used for systems for which "functional safety" is a mandatory requirement.

The use of RTOS code in safety-critical systems generally involves the use of code that has been rigorously tested and validated so that it conforms to one or more of the published safety standards. In the case of FreeRTOS, for example, there is an open source version of FreeRTOS and a validated version called SAFERTOS pre-certified to IEC 61508 for safety-critical applications. Currently, there is no pre-certified version of Zephyr RTOS. The Zephyr project is aiming to, eventually, be able to provide a version that has been certified for use in safety-critical applications. This is reflected in the Zephyr development and code review process.

Issues arising in the use of open source software in systems requiring functional safety include considerations such as those listed here:

- Open source software usually requires major transformation before it can be used.
- Mostly such transformation happens behind closed doors (if the license allows that).
- There may be a complete disconnect between original source and "certified" code.
- Transformation of open source code to be functionally safe is "expensive."
- Following standards very early in a project life cycle is a key factor.
- There are many standards dealing with safety-critical systems and software, and some members of this family are shown in the schematic partial family tree (Figure 1-1).

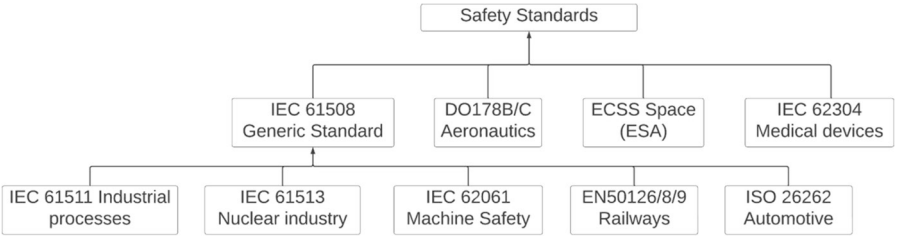

Figure 1-1. *Safety Standards, a partial family tree*

An example of going from an open source project to a system certified for use in safety-critical systems is FreeRTOS. SAFERTOS started with the functional model of the FreeRTOS kernel, but the kernel code was, then, redesigned, analyzed, and tested from a HAZOP perspective, and implemented according to an IEC 61508-3 SIL 3 development life cycle.

An ambition of the Zephyr RTOS initiative is to, eventually, provide an open source RTOS that can be used in safety-critical systems. Zephyr RTOS already provides many of the features expected of a safety-critical RTOS, but the real crux of the matter is the formal validation and testing of the system and its development process. The next few sections consider some of these issues.

Characteristics of an open source OS that would make it suitable for functional safety-oriented applications include the following:

- Open source implementation
- Small trusted code base (in terms of LoC)
- Safety-oriented architecture
- Built-in security model
- POSIX-compliant C library
- Support for deterministic thread scheduling
- Support for multi-core thread scheduling
- Proof that ISO-compliant development was done
- Accountability for the implementation
- Industry adoption
- Certification-friendly interfaces

The mission statement for Zephyr [1] is "to deliver the best-in-class RTOS for connected resource-constrained devices, built to be secure and safe." The Zephyr RTOS website contains presentations describing the various steps and approaches being followed that follow standard procedures for developing and testing safety-critical systems software. These include following the Verification and Validation aspects as formalized in the V-Model of software development. A useful discussion held during Open Source Summit Europe 2022 concerning these issues is worth viewing [2].

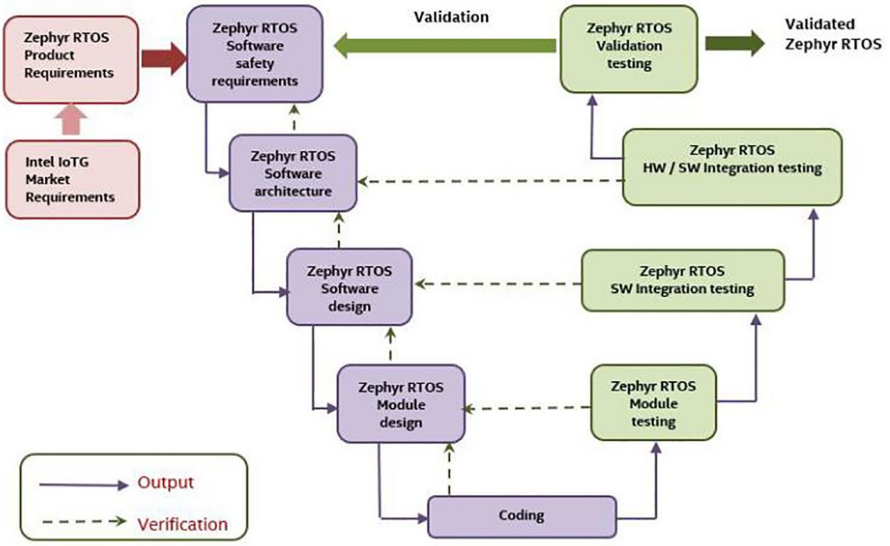

Figure 1-2. *Zephyr RTOS functional safety work products mapping to IEC 61508-3 V model [1]*

From the point of view of developing a safety-critical system quality RTOS, following the V-Model open source projects runs into issues such as the formal specification of features, producing comprehensive document, being able to produce traceability from requirements to source code, and being able to provide full information about the number of committers and information about them.

From the point of view of certification authorities, there is the problem that they are not familiar with open source development and there are no tried and tested methods for the certification of open source software.

Currently the standards being followed by Zephyr in regard to coding for Safety, Security, Portability, and Reliability in Embedded Systems are MISRA C:2012 (with Amendment 1, following MISRA C Compliance:2016 guidance) and the use of SEI CERT C and JPL (Jet Propulsion Laboratory

California Institute of Technology) as reference. As regards functional safety, the aim is to follow IEC 61508: 2010 (SIL 3 initially, eventually aiming to get to SIL 4). IEC 61508 is widely used by companies developing robotics systems and autonomous vehicles.

Writing embedded C code that conforms to MISRA guidelines is, these days, a widely accepted practice. Issues with MISRA and open source code that arise include the following:

- Some rules are very controversial; how to deal with those?
- Deciding which guidelines to deviate from and why
- MISRA C is proprietary; how can it be made more widely available?
- Finding the "open source" tools that check code and integrating these with CI

An example of a MISRA rule that is widely followed in embedded systems development is the following Rule 15.5 – A function should have a single point of exit at the end:

- Most readable structure
- Less likelihood of erroneously omitting function exit code
- Required by many safety standards
- IEC 61508
- ISO 26262

Reconciling Certification with Open Source

Reconciling an open source project with many potential contributors with a project that can produce safety-critical system certified software is tricky and represents "work in progress."

Various approaches are being explored and tried out. These include the following:

- Snapshotting a Source Tree (branch), validating it then controlling updates, which is a viable approach to software qualification.
- Defining the supported feature set as an up-front decision, bearing in mind that the more features that are supported, the greater the amount of documentation that will need to be provided and the amount of software testing that will need to be carried out. In this context, it will be important to automate as much of the information tracking as possible and to auto-generate documents from test and issue tracking systems.
- Obtaining proof-of-concept approval from a certification authority as early as possible.

An ideal project process that can combine the best aspects of open source development and critical system certification will be one based on a split development model having a flexible open instance path and an auditable instance path [3]. Aligning the auditable path with the open instance path will be dictated by the need to add new features and the costs of the certification process.

Zephyr As a Modular RTOS

The idea behind a modular RTOS is to develop it as a set of components that can be combined to be able to construct an application that incorporates only the functionality required for the application. This is not a new approach. The early versions of Microsoft's Windows NT operating system were modular with the possibility of being able to build operating system variants best suited to the task at hand.

Zephyr therefore tries to provide a solution to RTOS application development centered around a modular open source architecture appropriate for implementing a wide variety of use cases and design architectures running on connected, resource-constrained embedded controllers. Zephyr has an Apache 2.0 license, hosted at the Linux Foundation, and has extensive support for Bluetooth and for TCP/IP.

The modular aspects of the Zephyr OS can be conceptualized as a layered model shown in Figure 1-3 [3].

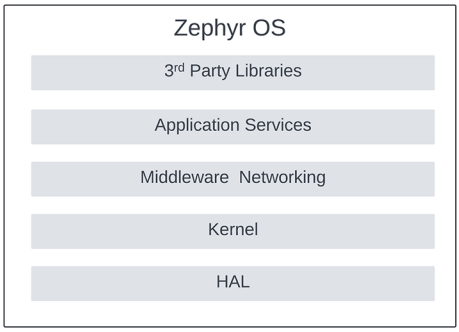

Figure 1-3. *Zephyr layered modular architecture*

Zephyr As a Fully Featured RTOS

An important aspect of Zephyr to be aware of is that Zephyr is not an ingredient – it provides a complete solution. Features supported by Zephyr include the following:

Safety features:

- Thread isolation
- Stack protection (HW/SW)
- Quality management (QM)
- Build time configuration
- No dynamic memory allocation
- Funtional SAfety (FuSA) (2019)

Security features:

- User-space support
- Crypto support
- Software updates

Configurable and modular kernel:

- Can configure the Zephyr kernel to run in 8K RAM
- Makes for scalable application code
- Only need to include what is required for the application

Cross-platform capabilities:

- Zephyr supports multiple architectures (ARM Cortex M, RISC-V, ARC, MIPS, Extensa).
- Native porting.
- Applications can be developed on Linux, Windows, and macOS platforms.