# 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 OS (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 when it is advisable to use an RTOS in an MCU application and when it might not be necessary at all.

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

- What is "real-time" anyway?
- Defining RTOS
- Deciding when to use an RTOS

## Technical requirements

There are no software or hardware requirements for this chapter.

## What is real-time anyway?

Any system that has a deterministic response to a given event can be considered "real-time." If a system is considered to *fail* when it doesn't meet a timing requirement, it must be real-time. How 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 also the severity of consequences if the required real-time deadlines are not met.

# The ranges of timing requirements

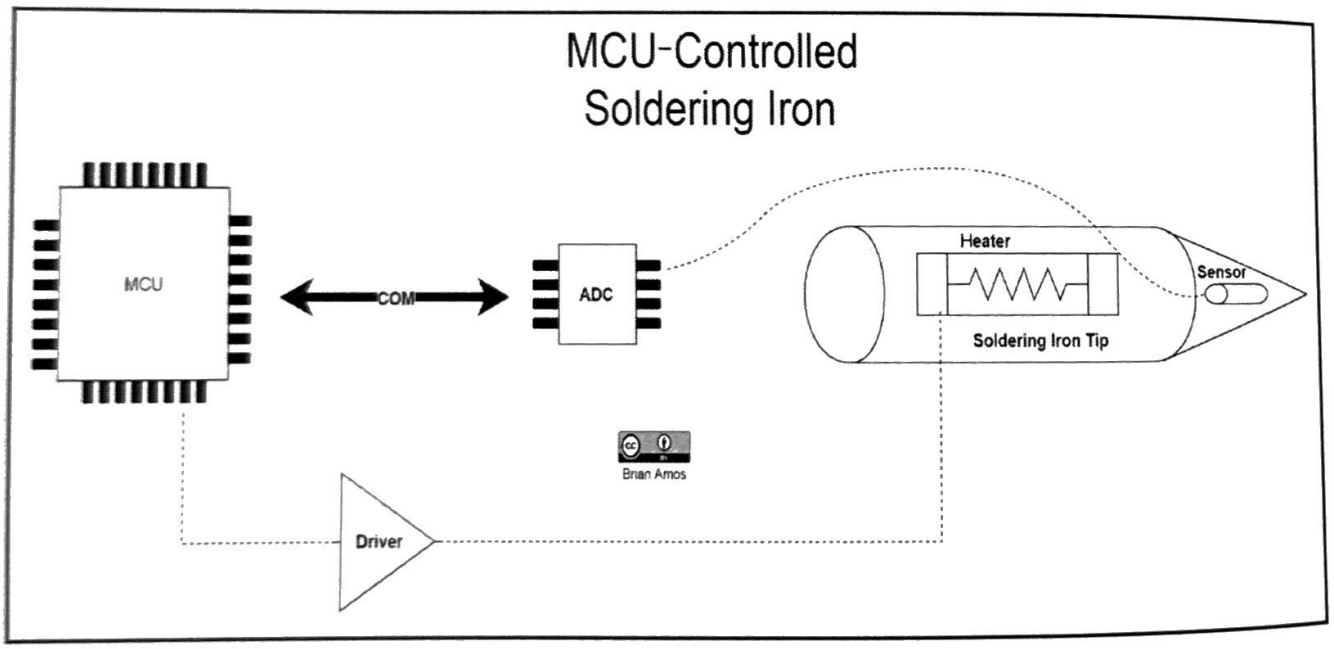
To illustrate the range of timing requirements that can be encountered, let's consider a few different systems that acquire readings from **analog-to-digital converters (ADCs)**.

The first system we'll look at is a control system that is set up to control the temperature of a soldering iron (as seen in the following diagram). The parts of the system we're concerned with are the MCU, ADC, sensor, and heater.

The MCU 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 diagram:




Since the temperature of the tip doesn't change incredibly 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) runs at an even slower pace, 5 Hz:

Now, on the other end of the ADC reading spectrum, we could have a high bandwidth network analyzer or oscilloscope that is going to be 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. A system like this requires huge amounts of processing to be performed and must adhere to extremely tight timing requirements, if it is to function properly.

Somewhere in the middle of the spectrum, you'll find systems such as closed-loop motion controllers, which will typically need to execute their 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.

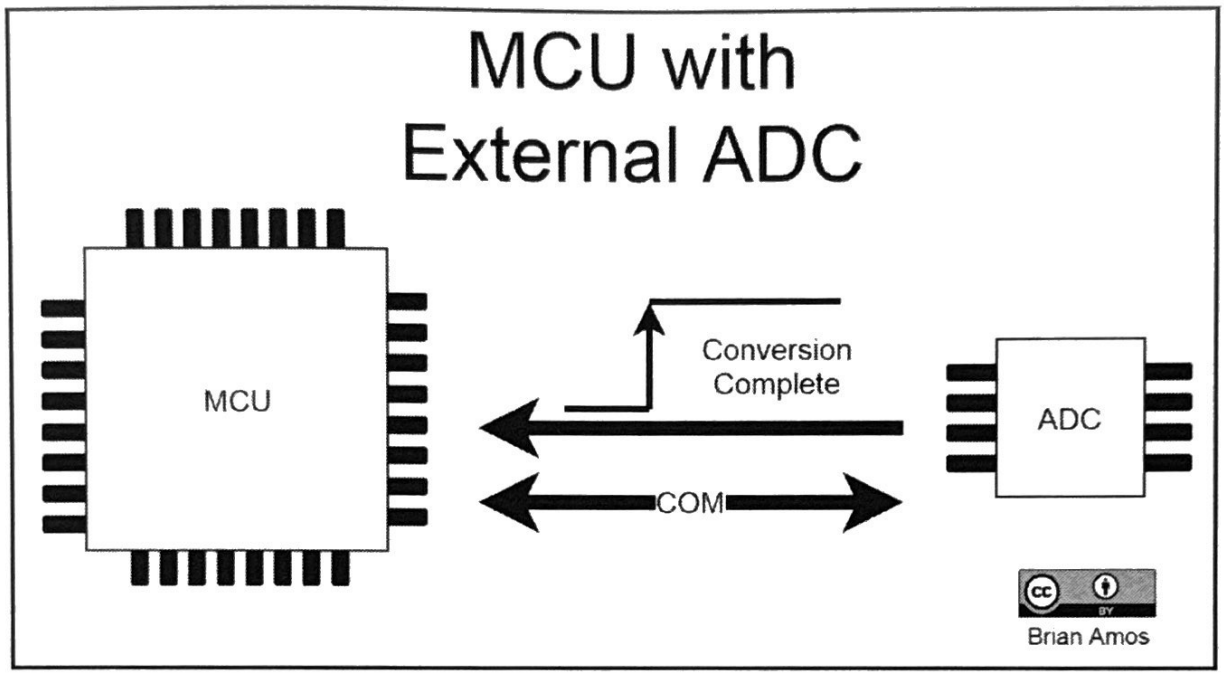
In some of the previous cases, such as the oscilloscope or soldering iron, failure to meet a timing requirement results in poor performance or incorrect data being reported. In the case of the soldering iron, this might be poor temperature control (which could cause damage to components). For the test equipment, missing deadlines could cause erroneous readings, which is a failure. This may not seem like a big deal to some people, but for the users of that equipment, who are relying on the accuracy of the data being reported, it is likely to matter a great deal. Some laboratory equipment that is used in standard verification provides checks for product conformance. If there is an undetected malfunction in the equipment that results in an inaccurate measurement, an incorrect value could be reported. It may be possible for a suspect test to be rerun. Eventually, however, if retesting is required too often and reliable readings can't be counted on, then the test equipment will start to become suspect and viewed as unreliable and sales will decline—all because a real-time requirement wasn't being consistently met.

In other systems, such as the flight control of a 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.

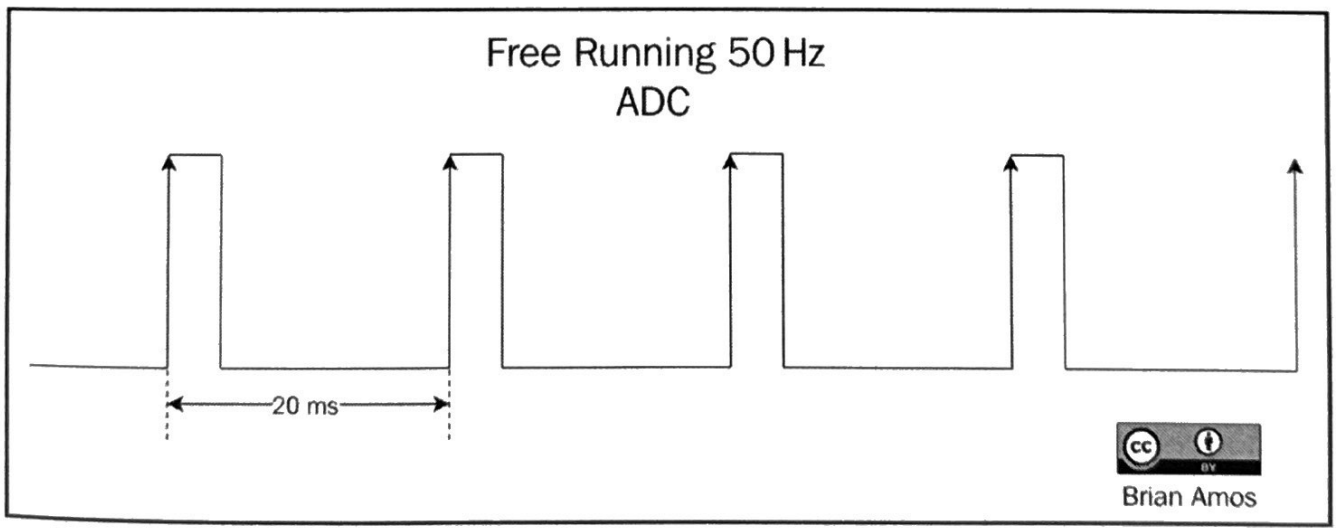
Thankfully, there are steps that can be taken to avoid all of these failure scenarios.

# The ways of guaranteeing real-time behavior

One of the easiest ways to ensure a system does what it is meant to do is to make sure it is as simple as possible while still meeting the requirements. 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.




The ADC will assert a hardware line, signaling a conversion has been completed and is ready for the MCU to transfer the reading to its internal memory. The MCU reading the ADC has up to 20 ms to transfer data from the ADC to internal memory before a new reading needs to be taken (as seen in the following diagram). 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:



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 isn't built *on top of* a preexisting kernel/scheduler of some type. Some engineers take this a step further, arguing that true bare-metal firmware can't use any preexisting libraries (such as vendor supply 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 fires. In this case, the only way for anything else to take control of the CPU is for the existing ISR to finish or for another higher-priority interrupt to fire.

Bare-metal firmware solutions 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.

As bare-metal implementations get to be more elaborate when dealing with events asynchronously, they start to overlap with 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—all of the RTOSes available today have been around for several years. The authors have been adapting and adding functionality the entire time to make them robust and flexible for different applications.

As programmers, if we come across a problem, we have a tendency to immediately reach for the nearest MCU and start coding. However, some functions of a product (especially true if a product has electro-mechanical components) are best handled without code at all. 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)**. If the functionality of the device is single-purposed, there are many cases where a full-blown RTOS can simply get in the way—so don't use one!

## Types of real-time systems

There are many different ways of achieving real-time behavior. The following section is a discussion on the various types of 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 a product, board, or even chip level (this approach is discussed in `Chapter 16`, *Multi-Processor and Multi-Core Systems*).

### Hardware

The original real-time system, hardware, is still the go-to for extremely tight tolerance and/or fast timing requirements. It can be implemented with discrete digital logic, analog components, programmable logic, or an **application-specific integrated component (ASIC)**. **Programmable logic devices (PLDs)**, **complex programmable logic devices (CPLDs)**, and **field-programmable gate arrays (FPGAs)** are the various members of the programmable logic device portion of this solution. 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.

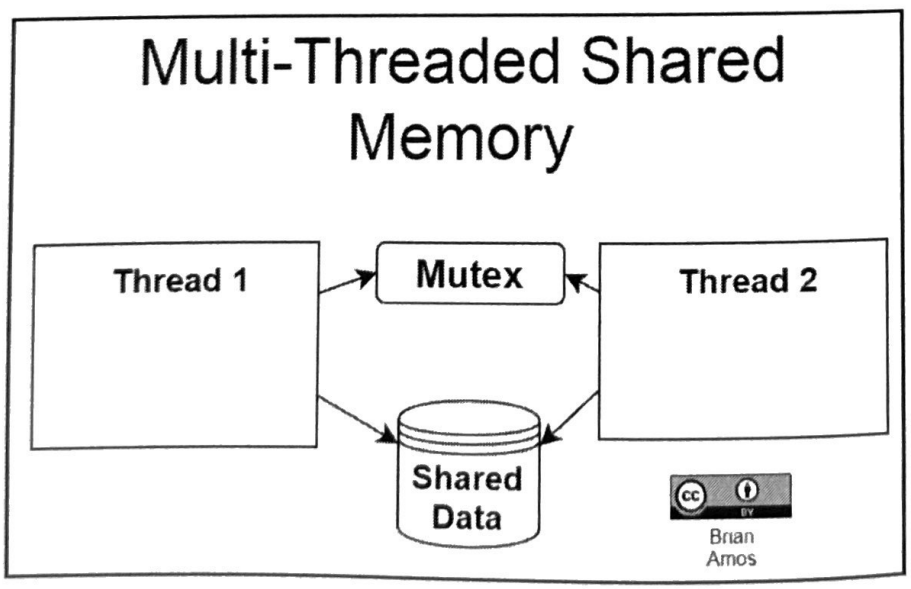
### Carefully crafted OS software

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). 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 real-time system.

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

# Defining RTOS

OSes (such as Windows, Linux, and macOS) were created as a way to provide a consistent programming environment that abstracted away the underlying hardware to make it easier to write and maintain computer programs. They provide the application programmer with many different *primitives* (such as threads and mutexes) that can be used to create more complex behavior. For example, it is possible to create a multi-threaded program that provides protected access to shared data:



## RTOS-based firmware

Firmware that runs a scheduling kernel on an MCU is RTOS-based firmware. The introduction of the scheduler and some RTOS-primitives allows tasks to operate under the illusion they have the processor to themselves (discussed in detail in `Chapter 2`, *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 all of these tasks running. Inter-dependencies can arise between tasks sharing data—if not handled properly, the dependency will cause a task to block unexpectedly. Although there are provisions for handling this, it does add 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. However, if handled improperly, this design paradigm can lead to more timing jitter and less determinism.

## RTOS-based software

Software running on a *full* OS that contains a **memory management unit** (**MMU**) and **central processing unit** (**CPU**) is considered RTOS-based software. Applications that are implemented with this approach can be highly complex, requiring many different interactions between various internal and external systems. The advantage of using a full OS is all of the capability that comes along with it—both hardware and software.

On the hardware side, there are generally more CPU cores available running at higher clock rates. There can be gigabytes 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. Underneath all of this capability 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 traditional OS. Because of this, getting deterministic performance is still within reach, just like with RTOS firmware.

## 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.

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 part 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), a few samples that are read slightly outside of their specified time are unlikely to completely destroy system control. If a control system has an ADC that automatically takes a new sample, if 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 or too frequently, the temperature stability will be ruined. In a particularly demanding system, it may only take a few missed samples before the entire control system is *out of spec*.

## 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 for keeping deadlines.

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.
- The 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 a time. In the case of a general-purpose OS, each thread will happily wait for the mutex to become available indefinitely before moving on to access the shared data. This is where RTOSes diverge from general-purpose OSes. 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, if **Thread 1** attempts to acquire **Mutex** and still doesn't have it after 100 ms, or 1 second, it will continue waiting for the mutex to become available.

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

Any OS that provides a deterministic way of executing a given piece of code can be considered a real-time OS. This definition of RTOS covers a fairly large number of systems.

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


Don't get too hung up on the differences between 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!

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

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). Another 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) that minimizes development 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 worth-while 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 manger 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, there are also open source and/or freely available middleware components available, some of which will be discussed 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 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 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 to transition to another RTOS, if the situation calls for it.

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. 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*.

# The range of RTOSes

RTOSes range in their functionality, as well as the architecture and size of the processor they're best suited to. On the smaller side, we have smaller 8–32-bit MCU-focused RTOSes, such as FreeRTOS, Keil RTX, Micrium µC, ThreadX, and many more. This class of RTOS is suitable for use on microcontrollers and provides a compact real-time kernel as the most basic offering. When moving from MCUs to 32- and 64-bit application processors, you'll tend to find RTOSes such as Wind River VxWorks and 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 USD$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 RTOSes provide a highly deterministic execution environment, they are often used in complex safety-critical applications. By *safety critical*, we generally mean a system whose failure could harm people or cause significant damage. These systems require deterministic operation because they must behave in a predictable way all the time. Guaranteeing 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 will typically either keep code for these systems extremely simple (so it is possible to prove mathematically that the system will function consistently and nothing can go wrong) or turn to a commercial RTOS solution, which has been through certification already, as a starting point. WITTENSTEIN SafeRTOS is a derivative of FreeRTOS that carries approvals for industrial, medical, and automotive use.

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 talk directly to 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.

# Deciding when to use 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 development time.

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 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, 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, **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 to one another, makes it very easy to argue that the additional complexity the RTOS brings will pay off. The RTOS helps us to ensure the lower priority, more complex tasks, such as networking and the filesystem stacks, won't interfere with the more time-critical tasks (such as controlling actuators and reading sensors). In many cases, there may be some form of control system that generally benefits from being run at well-defined intervals in time—a strength of the RTOS.

Now, consider a similar system 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 with an RTOS, but again, depending on the available team resources, this might be better left to a full-blown OS to handle (either RTOS or general-purpose), 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 their 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 to choose the solution that is best for your team and project at that point in time.

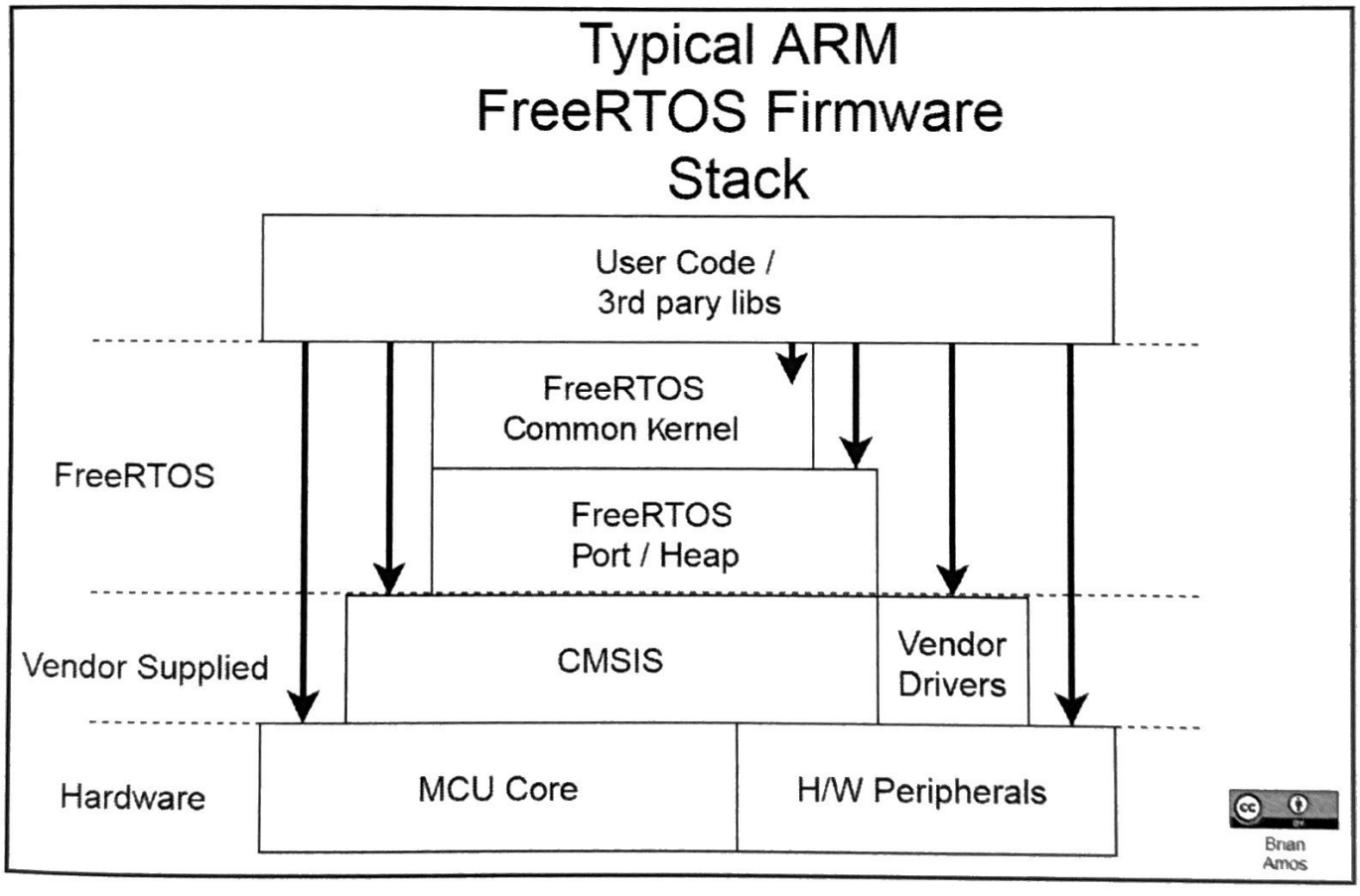
# Summary

In this chapter, we've covered how to identify real-time requirements, as well as the different platforms available for implementing real-time systems. At this point, you should have an appreciation for both the wide range of systems that can have real-time requirements, as well as the variety of ways there are to meet those real-time requirements.

In the next chapter, we'll start digging into MCU-based real-time firmware by taking a closer look at two different programming models—super loops and RTOS tasks.

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):




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, CMSIS, or raw hardware registers.