J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_5

5. Design Patterns

Jacob Beningo[1]
(1) Linden, MI, USA

I've worked on over 100 different embedded projects for the first 20 years of my career. These projects have ranged from low-power sensors used in the defense industry to safety-critical medical devices and flight software to children's toys. The one thing I have noticed between all these projects is that common patterns tend to repeat themselves from one embedded system to the next. Of course, this isn't an instance where a single system is designed, and we just build the same system repeatedly. However, having had the chance to review several dozen products where I had no input, these same patterns found their way into many systems. Therefore, they must contain some basic patterns necessary to build embedded systems.

This chapter will explore a few design patterns for embedded systems that leverage a microcontroller. We will not cover the well-established design patterns for object-oriented programming languages by the "Gang of Four (GoF),"[1] but instead focus on common patterns for building real-time embedded software from a design perspective. We will explore several areas such as single vs. multicore development, publish and subscribe models, RTOS patterns, handling interrupts, and designing for low power.

TipSchedule some time to read, or at least browse, *Design Patterns: Elements of Reusable Object-Oriented Software*.

Managing Peripheral Data

One of the primary design philosophies is that data dictates design. When designing embedded software, we must follow the data. In many cases, the data starts in the outside world, interacts with the system through a peripheral device,

and is then served up to the application. A major design decision that all architects encounter is "How to get the data from the peripheral to the application?". As it turns out, there are several different design mechanisms that we can use, such as

- Polling
- Interrupts
- Direct memory access (DMA)

Each of these design mechanisms, in turn, has several design patterns that can be used to ensure data loss is not encountered. Let's explore these mechanisms now.

Peripheral Polling

The most straightforward design mechanism to collect data from a peripheral is to simply have the application poll the peripheral periodically to see if any data is available to manage and process. Figure 5-1 demonstrates polling using a sequence diagram. We can see that the data becomes available but sits in the peripheral until the application gets around to requesting the data. There are several advantages and disadvantages to using polling in your design.

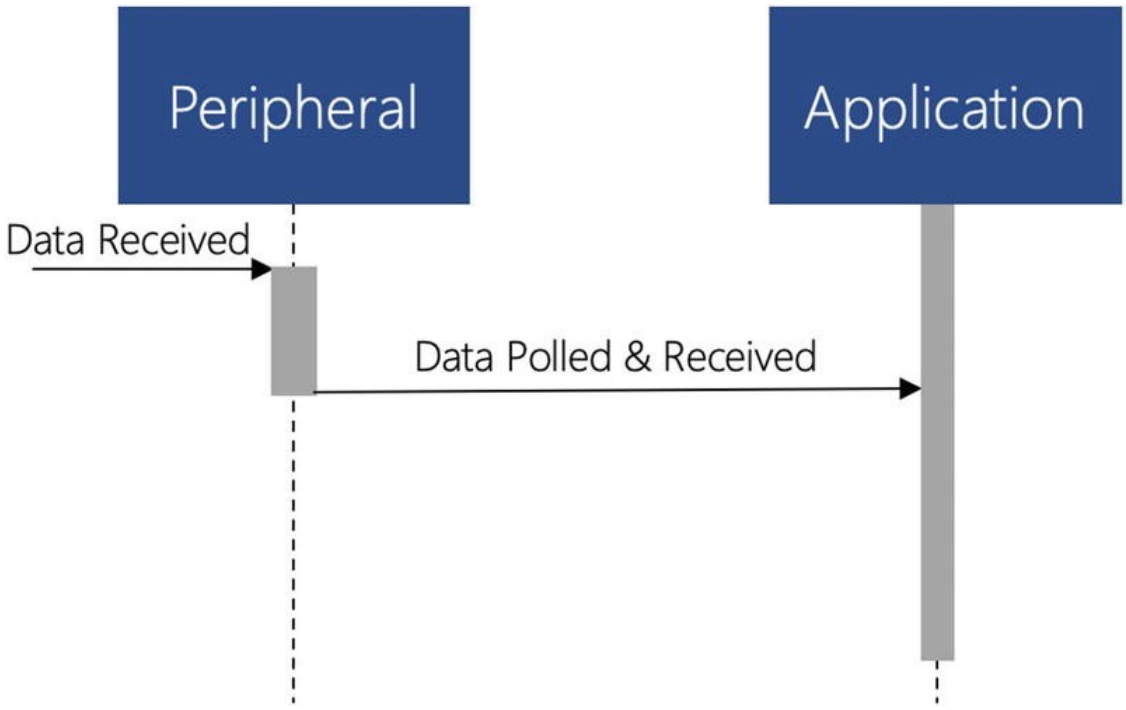

Figure 5-1 The application polls a peripheral for data on its schedule

The advantage of polling is that it is simple! There is no need to set up the interrupt controller or interrupt handlers for the peripheral. Typically, a

single application thread with very well-known timing is used to check a status bit within the peripheral to see if data is available or if the peripheral needs to be managed.

Unfortunately, there are more disadvantages to using polling than there are advantages. First, polling tends to waste processing cycles. Developers must allocate processor time to go out and check on the peripheral whether there is data there or not. In a resource-constrained or low-power system, these cycles can add up significantly. Second, there can be a lot of jitter and latency in processing the peripheral depending on how the developers implement their code.

For example, if developers decide that they are going to create a while loop that just sits and waits for data, they can get the data very consistently with low latency and jitter, but it comes at the cost of a lot of wasted CPU cycles. Waiting in this manner is polling using blocking. On the other hand, developers can instead not block and use a nonblocking method where another code is executed, but if the data arrives while another code is being executed, then there can be a delay in the application getting to the data adding latency. Furthermore, if the data comes in at nonperiodic rates, it's possible for the latency to vary, causing jitter in the processing time. The jitter may or may not affect other parts of the embedded system or cause system instability depending on the application.

Despite the disadvantages of polling, sometimes polling is just the best solution. If a system doesn't have much going on and it doesn't make sense to add the complexity of interrupts, then why add them? Debugging a system that uses interrupts is often much more complicated. If polling fits, then it might be the right solution; however, if the system needs to minimize response times and latency, or must wake up from low-power states, then interrupts might be a better solution.

Peripheral Interrupts

Interrupts are a fantastic tool available to designers and developers to overcome many disadvantages polling presents. Interrupts do precisely what their name implies; they interrupt the normal flow of the applica-

tion to allow an interrupt handler to run code to handle an event that has occurred in the system. For example, an interrupt might fire for a peripheral when data is available, has been received, or even transmitted. Figure 5-2 shows an example sequence diagram for what we can expect from an interrupt design.

The advantages of using an interrupt are severalfold. First, there is no need to waste CPU cycles checking to see if data is ready. Instead, an interrupt fires when there is data available. Next, the latency to get the data is deterministic. It takes the same number of clock cycles when the interrupt fires to enter and return from the interrupt service routine (ISR). The latency for a lower priority interrupt can vary though if a higher priority interrupt is running or interrupts it during execution. Finally, jitter is minimized and only occurs if multiple interrupts are firing simultaneously. In this case, the interrupt with the highest priority gets executed first. The jitter can potentially become worse as well if the interrupt fires when an instruction is executing that can't be interrupted.

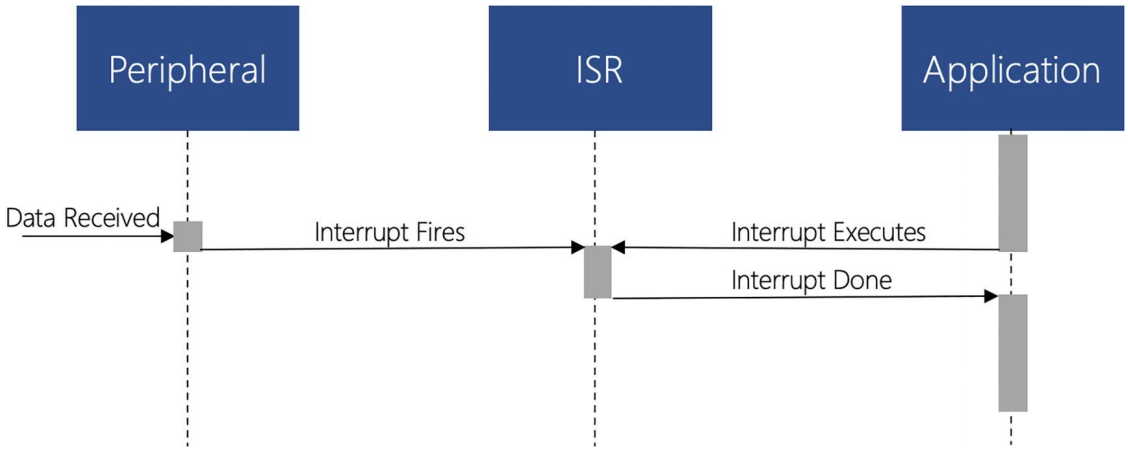

Figure 5-2 When data is received in the peripheral, it fires an interrupt which stops executing the application, runs the interrupt handler, and then returns to running the application

Despite interrupts solving many problems associated with polling, there are still some disadvantages to using interrupts. First, interrupts can be complicated to set up. While interrupt usage increases complexity, the benefits usually overrule this disadvantage. Next, designers must be careful not to use interrupts that fire too frequently. For example, trying to use interrupts to debounce a switch can cause an interrupt to fire very frequently, potentially starving the main application and breaking its real-time performance. Finally, when interrupts are used to receive data, developers must carefully manage what they do in the ISR. Every clock

cycle spent in the ISR is a clock cycle away from the application. As a result, developers often need to use the ISR to handle the immediate action required and then offload processing and non-urgent activities to the application, causing software design complexity to increase.

Interrupt Design Patterns

When an interrupt is used in a design, there is a chance that the work performed on the data will take too long to run in the ISR. When we design an ISR, we want the interrupt to

- Run as quickly as possible (to minimize the interruption)
- Avoid memory allocation operations like declaring nonstatic variables, manipulating the stack, or using dynamic memory
- Minimize function calls to avoid clock cycle overhead and issues with nonreentrant functions or functions that may block.

There's also a good chance that the data just received needs to be combined with past or future data to be useful. We can't do all those operations in a timely manner within an interrupt. We are much better served by saving the data and notifying the application that data is ready to be processed. When this happens, we need to reach for design patterns that allow us to get the data quickly, store it, and get back to the main application as soon as possible.

Designers can leverage several such patterns used on bare-metal and RTOS-based systems. A few of the most exciting patterns include

- Linear data store
- Ping-pong buffers
- Circular buffers
- Circular buffer with semaphores
- Circular buffer with event flags
- Message queues

Linear Data Store Design Pattern

A linear data store is a shared memory location that an interrupt service routine

can directly access, typically to write new data to memory. The application code, usually the data reader, can also directly access this memory, as shown in Figure 5-3.

Figure 5-3 An interrupt (writer) and application code (reader) can directly access the data store

Now, if you've been designing and developing embedded software for any period, you'll realize that linear data stores can be dangerous! Linear data stores are where we often encounter race conditions because access to the data store needs to be carefully managed so that the ISR and application aren't trying to read and write from the data store simultaneously. In addition, the variables used to share the data stored between the application and the ISR also need to be declared volatile to prevent the compiler from optimizing out important instructions caused by the interruptible nature of the operations.

Data stores often require the designer to build mutual exclusion into the data store. Mutual exclusion is needed because data stores have a critical section where if the application is partway through reading the data when an interrupt fires and changes it, the application can end up with corrupt data. We don't care how the developers implement the mutex at the design level, but we need to make them aware that the mutex exists. I often do this by putting a circular symbol on the data store containing either an "M" for a mutex or a key symbol, as shown in Figure 5-4. Unfortunately, at this time, there are no standards that support official nomenclature for representing a mutex.

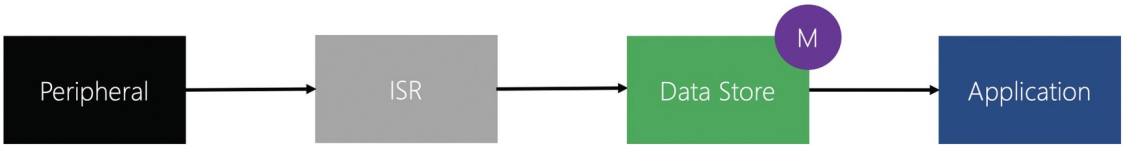

Figure 5-4 The data store must be protected by a mutex to prevent race conditions. The mutex is shown in the figure by an "M"

Ping-Pong Buffer Design Pattern

Ping-pong buffers, also sometimes referred to as double buffers, offer another design solution meant to help alleviate some of the race condition problems

encountered with a data store. Instead of having a single data store, we have two identical data stores, as shown in Figure 5-5.

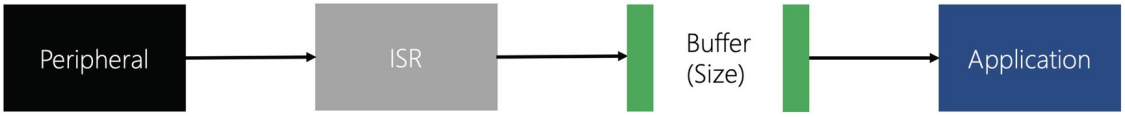

Figure 5-5 A ping-pong buffer transfers data between the ISR and application

Now, at first, having two data stores might seem like an opportunity just to double the trouble, but it's a potential race condition saver. A ping-pong buffer is so named because the data buffers are used back and forth in a ping-pong-like manner. For example, at the start of an application, both buffers are marked as write only – the ISR stores data in the first data store when data comes in. When the ISR is done and ready for the application code to read, it marks that data store as ready to read. While the application reads that data, the ISR stores it in the second data store if additional data comes in. The process then repeats.

Circular Buffer Design Pattern

One of the simplest and most used patterns to get and use data from an interrupt is to leverage a circular buffer. A circular buffer is a data structure that uses a single, fixed-size buffer as if it were connected end to end. Circular buffers are often represented as a ring, as shown in Figure 5-6. Microcontroller memory is not circular but linear. When we build a circular buffer in code, we specify the start and stop addresses, and once the stop address is reached, we loop back to the starting address.

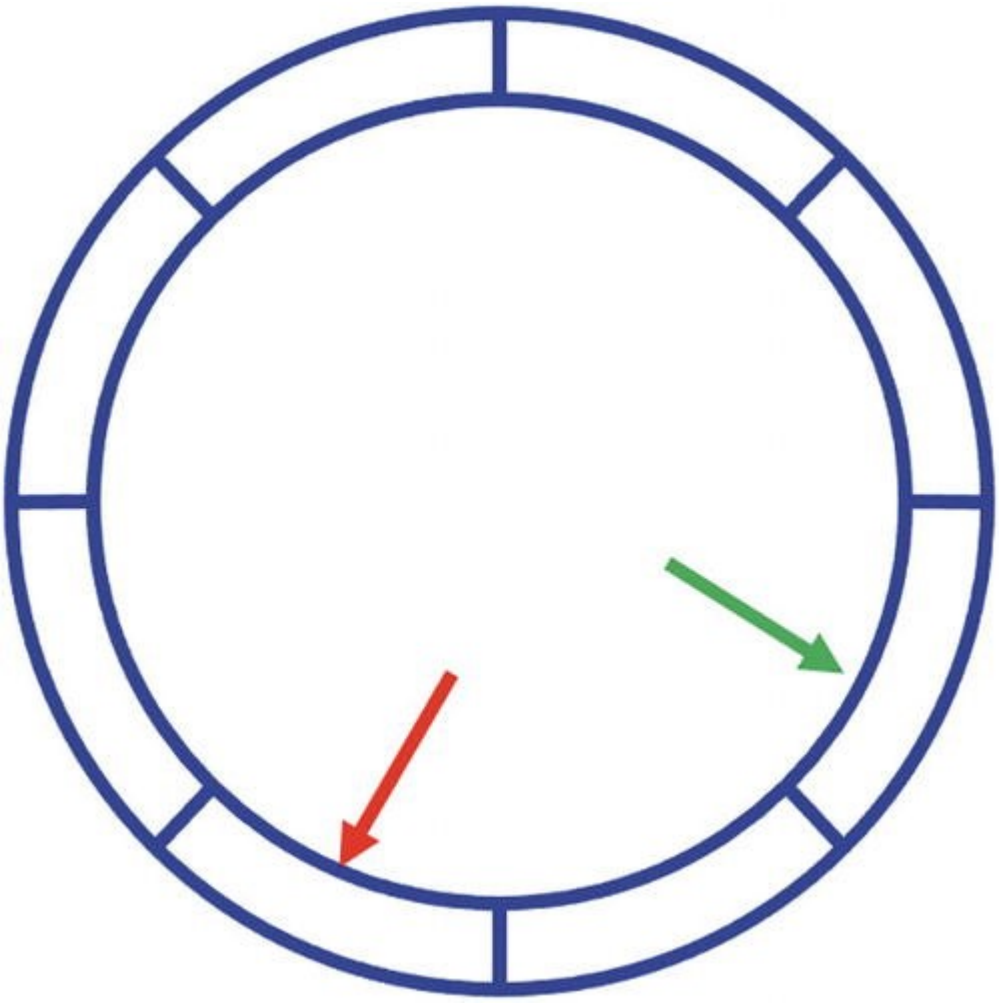

Figure 5-6 An eight-element circular buffer representation. The red arrow indicates the head where new data is stored. The green arrow represents the tail, where data is read out of the buffer

The idea with the circular buffer is that the real-time data we receive in the interrupt can be removed from the peripheral and stored in a circular buffer. As a result, the interrupt can run as fast as possible while allowing the application code to process the circular buffer at its discretion. Using a circular buffer helps ensure that data is not lost, the interrupt is fast, and we still process the data reasonably.[2]

The most straightforward design pattern for a circular buffer can be seen in Figure 5-7. In this pattern, we are simply showing how data moves from the peripheral to the application. The data starts in the peripheral, is handled by the ISR, and is stored in a circular buffer. The application can come and retrieve data from the circular buffer when it wants to. Of course, the circular buffer needs to be sized appropriately, so the buffer does not overflow.

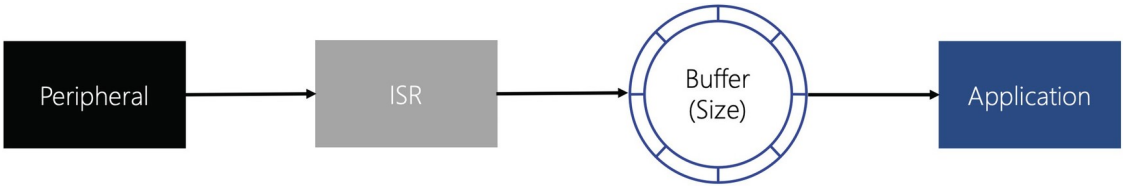

Figure 5-7 The data flow diagram for moving data from the peripheral memory storage into the application where it is used

Circular Buffer with Notification Design Pattern

The circular buffer design pattern is great, but there is one problem with it that we haven't discussed; the application needs to poll the buffer to see if there is new data available. While this is not a world-ending catastrophe, it would be nice to have the application notified that the data buffer should be checked. Two methods can signal the application: a semaphore and an event flag.

A semaphore is a synchronization primitive that is included in real-time operating systems. Semaphores can be used to signal tasks about events that have occurred in the application. We can leverage this tool in our design pattern, as shown in Figure 5-8. The goal is to have the ISR respond to the peripheral as soon as possible, so there is no data loss. The ISR then saves the data to the circular buffer. At this point, the application doesn't know that there is data to be processed in the circular buffer without polling it. The ISR then signals the application by giving a semaphore before completing execution. When the application code runs, the task that manages the circular buffer can be unblocked by receiving the semaphore. The task then processes the data stored in the circular buffer.

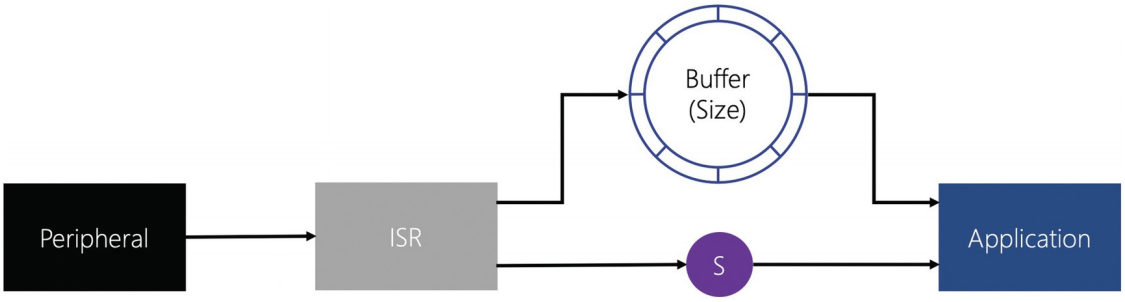

Figure 5-8 A ring buffer stores incoming data with a semaphore to notify the application that data is available for processing

Using a semaphore is not the only method to signal the application that data is ready to be processed. Another approach is to replace the semaphore with an event flag. An event flag is an individual bit that is usually

part of an event flag group that signals when an event has occurred. Using an event flag is more efficient in most real-time operating systems than using a semaphore. For example, a designer can have 32 event flags in a single RAM location on an Arm Cortex-M processor. In contrast, just a single semaphore with its semaphore control block can easily be a few hundred bytes.

Semaphores are often overused in RTOS applications because developers jump straight into the coding and often don't take a high-level view of the application. The result is a bunch of semaphores scattered throughout the system. I've also found that developers are less comfortable with event flags because they aren't covered or discussed as often in classes or engineering literature.

An example design pattern for using event flags and interrupts can be seen in Figure 5-9. We can represent an event flag version of a circular buffer with a notification design pattern. As you can see, the pattern itself does not change, just the tool we use to implement it. The implementation here results in fewer clock cycles being used and less RAM.

TipSemaphores are often overused in RTOS applications. Consider first using an event flag group, especially for event signals with a binary representation.

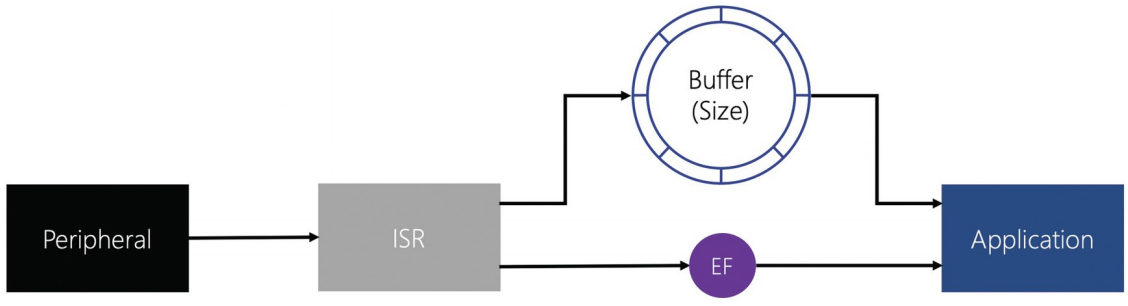

Figure 5-9 A ring buffer stores incoming data with an event flag to notify the application that data is available for processing

Message Queue Design Pattern

The last method we will look at for moving data from an interrupt into the application uses message queues. Message queues are a tool available

in real-time operating systems that can take data of a preset set maximum size and queue it up for processing by a task. A message queue can typically store more than a single message and is configurable by the developer.

To leverage the message queue design pattern, the peripheral once again produces data retrieved by an ISR. The ISR then passes the data into a message queue that can be used to signal an application task that data is available for processing. When the application task has the highest priority, the task will run and process the stored data in the queue. The overall pattern can be seen in Figure 5-10.

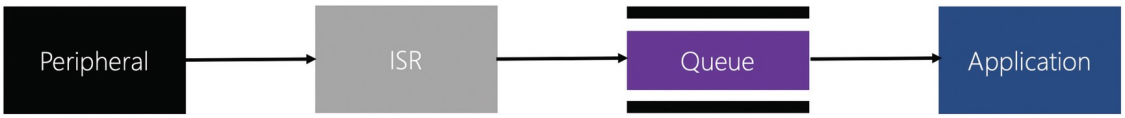

Figure 5-10 A message queue stores incoming data and passes it into the application for processing

Using a message queue is like using a linear buffer with a semaphore. While designers may be tempted to jump immediately to using a message queue, it's essential to consider the implications carefully. A message queue typically requires more RAM, ROM, and processing power than the other design patterns we discussed. However, it can also be convenient if there is enough horsepower and the system is not a highly constrained embedded system.

Direct Memory Access (DMA)

Many microcontrollers today include a direct memory access (DMA) controller that allows individual channels to be set up to move data in the following ways without interaction from the CPU:

- RAM to RAM
- Peripheral to RAM
- Peripheral to peripheral

A typical block diagram for DMA and how it interacts with the CPU, memory, and peripherals can be seen in Figure 5-11.

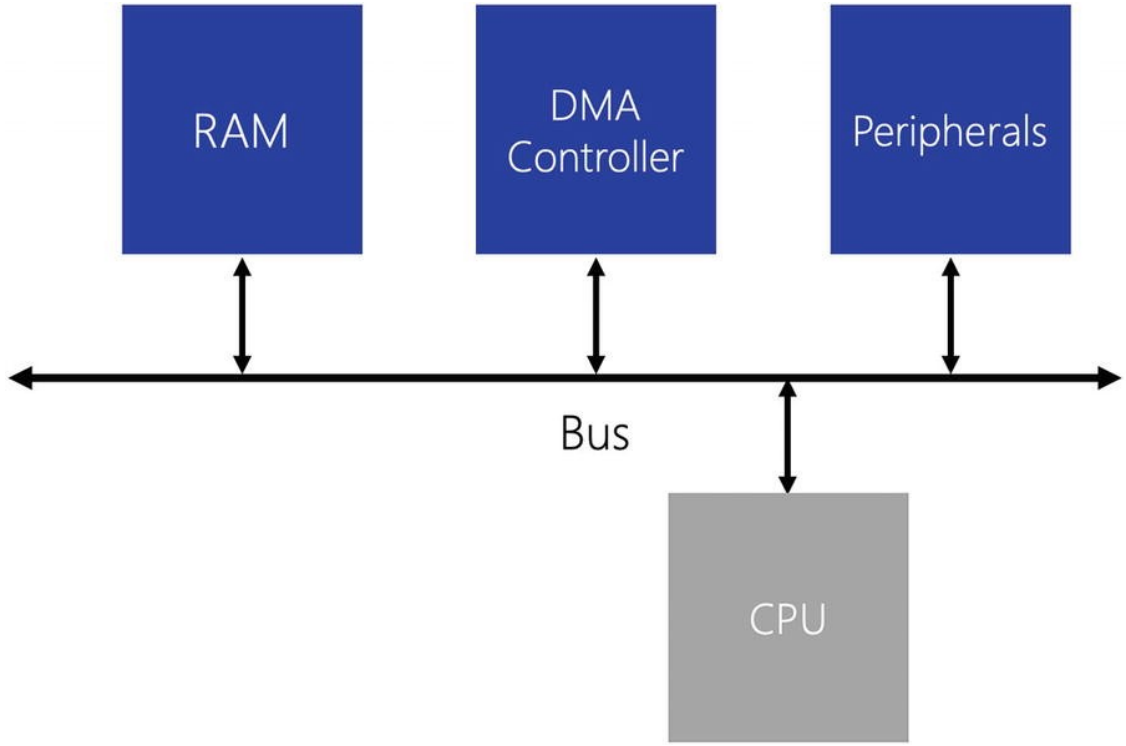

Figure 5-11 The DMA controller can transfer data between and within the RAM and peripherals without the CPU

The DMA controller can help to dramatically improve data throughput between the peripheral and the application. In addition, the DMA controller can be leveraged to alleviate the CPU from having to run ISRs to transfer data and minimize wasted compute cycles. For example, one design pattern that is especially common in analog-to-digital converters is to leverage DMA to perform and transfer all the converted analog channels and then copy them to a buffer.

For example, Figure 5-12 shows a design pattern where the DMA controller transfers peripheral data into a circular buffer. After a prespecified number of byte transfers, the DMA controller will trigger an interrupt. The interrupt, in this case, uses a semaphore to signal the application that there is data ready to be processed. Note that we could have used one of the other interrupt patterns, like an event flag, but I think this gives you the idea of how to construct different design patterns.

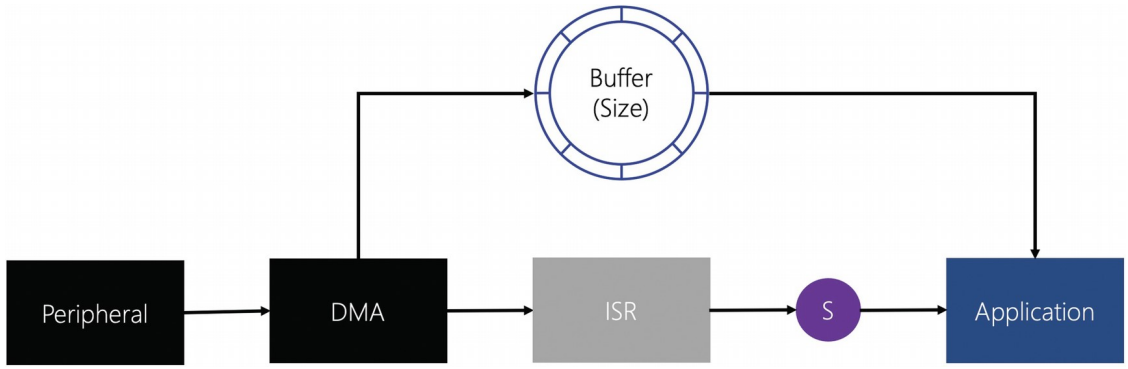

Figure 5-12 An example design pattern that uses the DMA controller to move data from a peripheral into a circular buffer

Again, the advantage is using the DMA controller to move data without the CPU being involved. The CPU is able to go execute other instructions while the DMA controller is performing transfer operations. Developers do have to be careful that they don't access or try to manipulate the memory locations used by the DMA controller while a transfer is in progress. A DMA interrupt is often used to signal that it is safe to operate on the data. Don't forget that it is possible for the DMA to be interrupted.

RTOS Application Design Patterns

Design patterns can cover nearly every aspect of the embedded software design process, including the RTOS application. For example, in an RTOS application, designers often need to synchronize the execution of various tasks and data flowing through the application. Two types of synchronization are usually found in RTOS applications, resource synchronization and activity synchronization.

Resource synchronization determines whether the access to a shared resource is safe and, if not, when it will be safe.[3] When multiple tasks are trying to access the same resource, the tasks must be synchronized to ensure integrity. Activity synchronization determines whether the execution has reached a specific state and, if not, how to wait and notify that the state has been reached.[4] Activity synchronization ensures correct execution order among cooperating tasks.

Resource synchronization and activity synchronization design patterns come in many shapes and sizes. Let's explore a few common synchroniza-

tion patterns in more detail.

Resource Synchronization

Resource synchronization is about ensuring multiple tasks, or tasks and interrupts, that need to access a resource, like a memory location, do so in a coordinated manner that avoids race conditions and memory corruption. For example, let's say we are working on a system that acquires sensor data and uses that data to control a system output. The system might contain two tasks, a sensor task and a control task, in addition to a shared memory location (RAM) that stores the sensor data to be displayed, as shown in Figure 5-13.

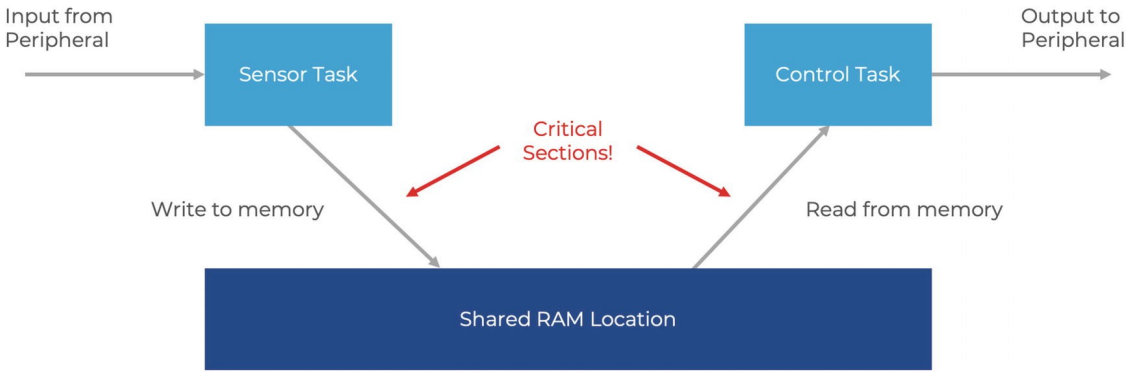

Figure 5-13 Two tasks looking to access the same memory location require a design pattern to synchronize resource accesses[5]

In Figure 5-13, you can see that the sensor task acquires data from a device and then writes it to memory. The control task reads the data from memory and then uses it to generate an output. Marked in red, you can see that the write and read operations are critical sections! If the control task is in the process of reading the memory when the sensor task decides to update the memory, we can end up with data corruption and perhaps a wrong value being used to control a motor or other device. The result could be really bad things happening to a user!

There are three ways designers can deal with resource synchronization: interrupt locking, preemption locking, and mutex locking.

> **Tip**A good architect will minimize the need for resource synchronization. Avoid it if possible, and when necessary, use these techniques (with mutex locking being the preference).

Interrupt Locking

Interrupt locking occurs when a system task disables interrupts to provide resource synchronization between the task and an interrupt. For example, suppose an interrupt is used to gather sensor data that is written to the shared memory location. In that case, the control task can lock (disable) the specific interrupt during the reading process to ensure that the data is not changed during the read operation, as shown in Figure 5-14.

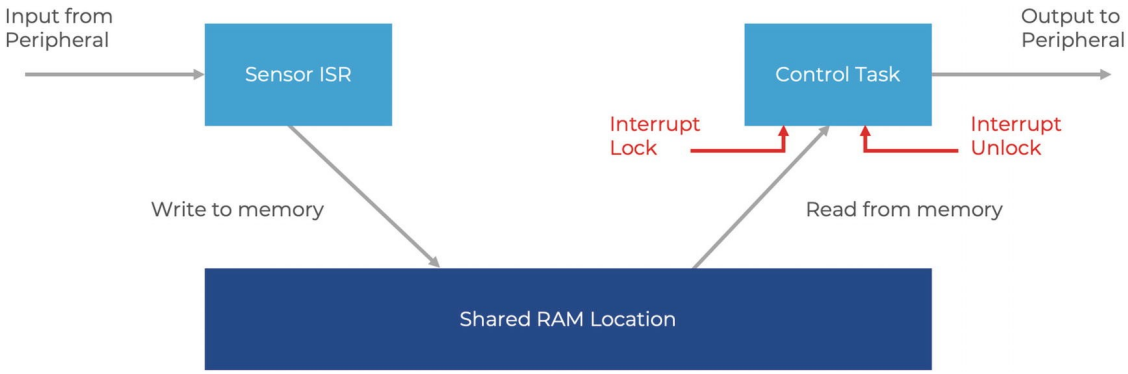

Figure 5-14 Interrupt locking is used to disable interrupts to prevent a race condition with the sensor task

Interrupt locking can be helpful but can cause many problems in a real-time embedded system. For example, by disabling the interrupt during the read operation, the interrupt may be missed or, best case, delayed from execution. Delayed execution can add unwanted latency or even jitter into the system.

Preemption Lock

Preemption lock is a method that can be used to ensure that a task is uninterrupted during the execution of a critical section. Preemption lock temporarily disables the RTOS kernel preemptive scheduler during the critical section, as shown in Figure 5-15.

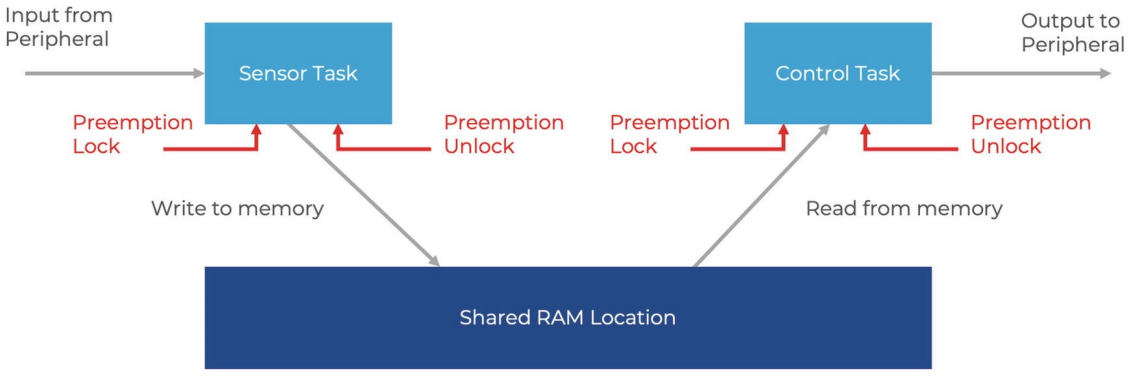

Figure 5-15 A preemption lock protects the critical section by not allowing the RTOS kernel to preempt

the executing task

Preemption lock is a better technique than interrupt locking because critical system interrupts are still allowed to run; however, a higher priority task may be delayed in its execution. Preemption lock can also introduce additional latency and jitter into the system. There is also a possibility that a higher priority task execution could be delayed due to the kernel's preemptive scheduler being disabled.

Mutex Lock

One of the safest and most recommended methods for protecting a shared resource is to use a mutex lock. A mutex is an RTOS object whose sole purpose is to provide mutual exclusion to shared resources, as shown in Figure 5-16.

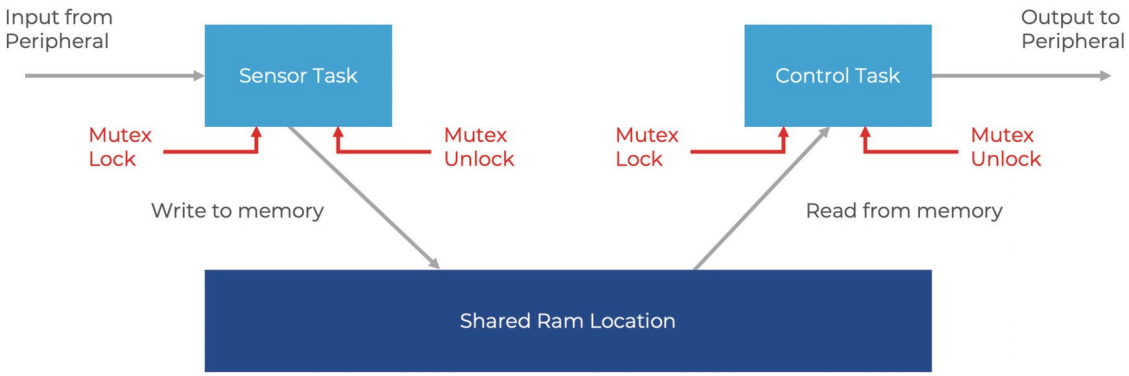

Figure 5-16 A mutex lock is used to protect a critical section by creating an object whose state can be checked to determine whether it is safe to access the shared resource

A mutex will not disable interrupts. It will not disable the kernel's preemptive scheduler. It will protect the shared resource in question. One potential problem with a mutex lock protecting a shared resource is that developers need to know it exists! For example, if a developer wrote a third task that would access the sensor data, if they did not know it was a shared resource, they could still just directly access the memory! A mutex is an object that manages a lock state, but it does not physically lock the shared resource.

I highly recommend that if you have a shared resource with which you would like to use a mutex lock, either encapsulate the mutex behind the scenes in the code module or build the mutex into your data structure.

Then, when the mutex is built into the data structure when developers use dot notation to access the structure members, the mutex will show up, and they'll realize that the resource is shared and that they need to check to lock the mutex first. (My preference is to encapsulate the module's behavior, but it depends on the use case.)

Activity Synchronization

Activity synchronization is all about coordinating task execution. For example, let's say we are working on a system that acquires sensor data and uses the sensor values to drive a motor. We most likely would want to signal the motor task that new data is available so that the task doesn't act on old, stale data. Many activity synchronization patterns can be used to coordinate task execution when using a real-time operating system. In this section, we are going to look at just a few.[6]

Unilateral Rendezvous (Task to Task)

The unilateral rendezvous is the first method we will discuss to synchronize two tasks. The unilateral rendezvous uses a binary semaphore or an event flag to synchronize the tasks. For example, Figure 5-17 shows tasks 1 and 2 are synchronized with a unilateral rendezvous. First, task 2 executes its code to a certain point and then becomes blocked. Task 2 will remain blocked until task 1 reaches the point where it is ready for task 2 to resume executing. Then, task 1 notifies task 2 that it's okay to proceed by giving a semaphore. Task 2 then unblocks, takes the semaphore, and continues to execute.

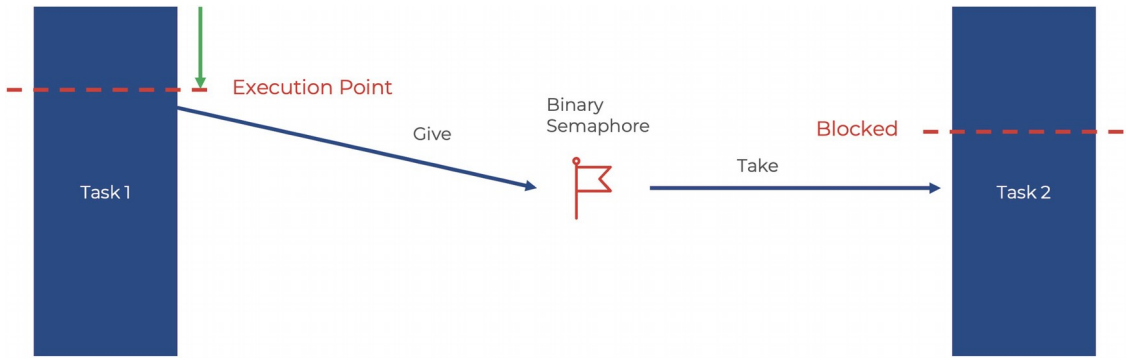

Figure 5-17 An example of unilateral rendezvous synchronization between tasks using a binary semaphore

Unilateral Rendezvous (Interrupt to Task)

The unilateral rendezvous is not only for synchronizing two tasks together. Unilateral rendezvous can also synchronize and coordinate task execution between an interrupt and a task, as shown in Figure 5-18. The difference here is that after the ISR gives the semaphore or the event flag, the ISR will continue to execute until it is complete. In addition, the unilateral rendezvous between two tasks may cause the task that gives the semaphore to be preempted by the other task if the second task has a higher priority.

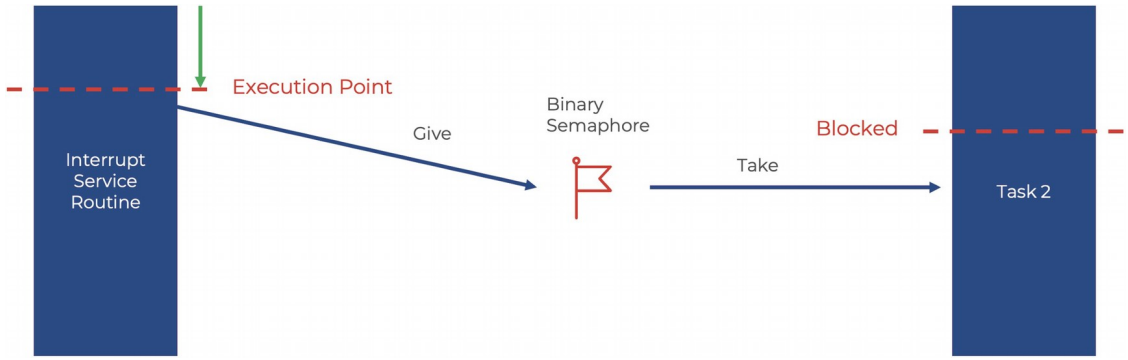

Figure 5-18 A unilateral rendezvous can be used to synchronize an ISR and task code

Unilateral rendezvous can also be used with counting semaphores to create a credit tracking design pattern, as shown in Figure 5-19. In this pattern, multiple interrupts might fire before task 2 can process them. In this case, a counting semaphore is used to track how many times the interrupt fired. An example could be an interrupt that saves binary byte data to a buffer. To track how many bytes are available for processing, the interrupt each time increments the counting semaphore.

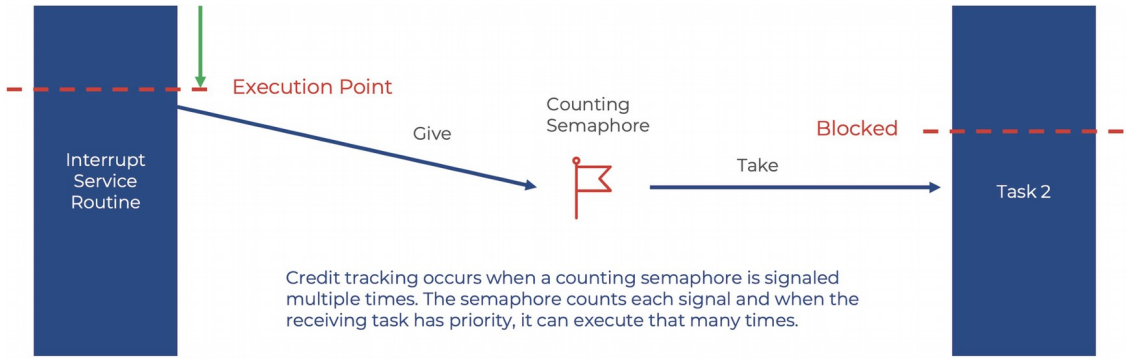

Figure 5-19 A counting semaphore is used in this unilateral rendezvous to track how many “credits” have been provided by the receiving task to act upon

Bilateral Rendezvous

In case you didn’t notice, a unilateral rendezvous is a design pattern for synchronizing tasks where one task needs to coordinate with the second task in

one direction. On the other hand, a bilateral rendezvous may be necessary if two tasks need to coordinate in both directions between them. An example of bilateral rendezvous can be seen in Figure 5-20.

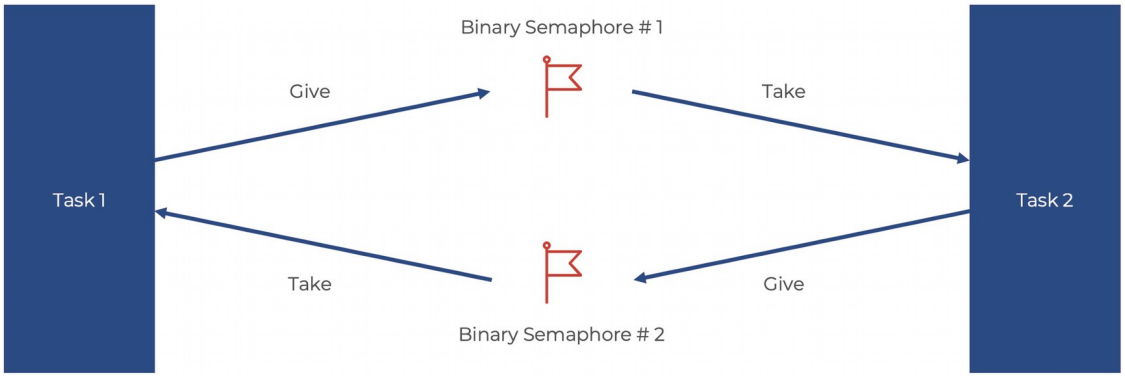

Figure 5-20 A bilateral rendezvous that uses two binary semaphores to synchronize execution between two tasks

In Figure 5-20, task 1 reaches a certain point in its execution and then gives a semaphore to task 2. Task 1 might continue to execute to a specific point where it then blocks and waits to receive the second semaphore. The exact details depend on the application's needs and how the tasks are designed. For example, task 2 might wait for the first semaphore and then, upon taking it, runs some code until it reaches its synchronization point and then gives the second binary semaphore to task 1.

Synchronizing Multiple Tasks

Sometimes, a design will reach a complexity level where multiple tasks need to synchronize and coordinate their execution. The exact design pattern used will depend on the specific application needs. This book will only explore a straightforward design pattern, the broadcast design pattern.

The broadcast design pattern allows multiple tasks to block until a semaphore is given, an event flag occurs, or even a message is placed into a message queue. Figure 5-21 shows an example of a task or an interrupt giving a semaphore broadcast to three other tasks. Each receiving task can consume the semaphore and execute its task code once it is the highest priority task ready to run.

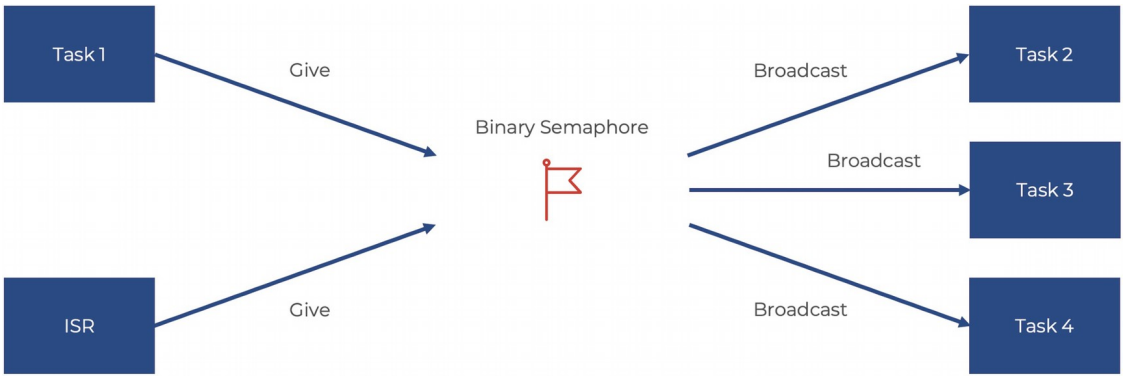

Figure 5-21 A task or an ISR can give a binary semaphore broadcast consumed by multiple tasks

Broadcasting can be a useful design pattern, but it is essential to realize that it may not be implemented in all real-time operating systems. If you are interested in triggering more than one task off an event, you'll have to check your RTOS documentation to see if it can do so. If broadcasting is not supported, you can instead create multiple semaphores given by the task or the interrupt. Using multiple semaphores isn't as elegant from a design standpoint or efficient from an implementation standpoint, but sometimes that is just how things go in software.

Publish and Subscribe Models

Any developer working with embedded systems in the IoT industry and connecting to cloud services is probably familiar with the publish/subscribe model. In many cases, an IoT device will power up, connect to the cloud, and then subscribe to message topics it wants to receive. The device may even publish specific topics as well. Interestingly, even an embedded system running FreeRTOS can leverage the publish and subscribe model for its embedded architecture.

The general concept, shown in Figure 5-22, is that a broker is used to receive and send messages below to topics. Publishers send messages to the broker specifying which topic the message belongs to. The broker then routes those messages to subscribers who request that they receive messages from the specific topics. It's possible that there would be no subscribers to a topic, or that one subscriber subscribes to multiple topics, and so forth. The publish and subscribe pattern is excellent for abstracting the system architecture. The publisher doesn't know who listens to its messages and doesn't care. The subscriber only cares about messages that it is subscribed to. The result is a scalable system.

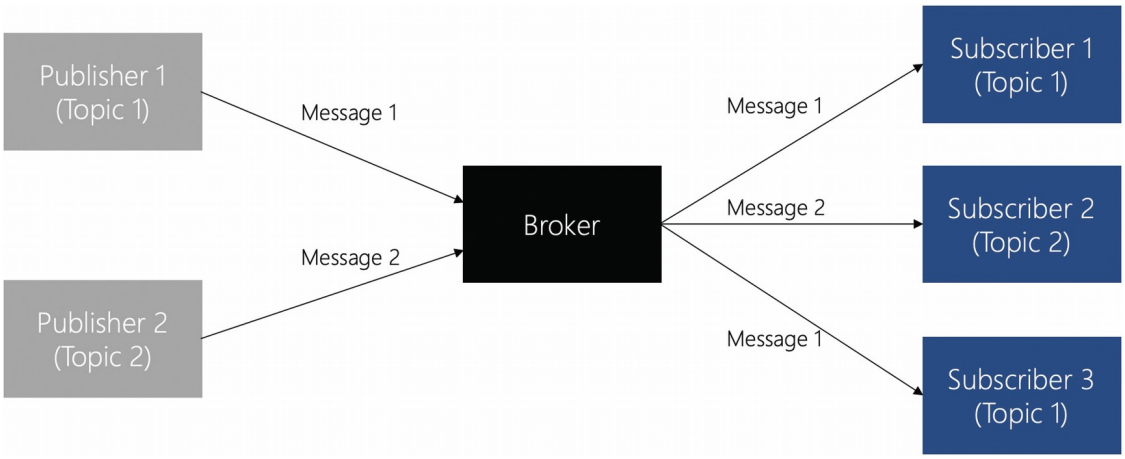

Figure 5-22 An example publish/subscribe system where two publishers publish messages to the broker, which then routes the messages to the subscribers of the message topics

Let's, for a moment, think about an example. Let's say we have a system that will collect data from several sensors. That sensor data is going to be stored in nonvolatile memory. In addition, that data needs to be transmitted as part of a telemetry beacon for the system. On top of that, the data needs to be used to maintain the system's orientation through use in a control loop.

We could architect this part of the system in several ways, but one exciting way would be to use the publish and subscribe design pattern. For example, we could add a message broker to our system, define the sensor acquisition task as a publisher, and then have several subscribers that receive data messages and act on that data, as shown in Figure 5-23.

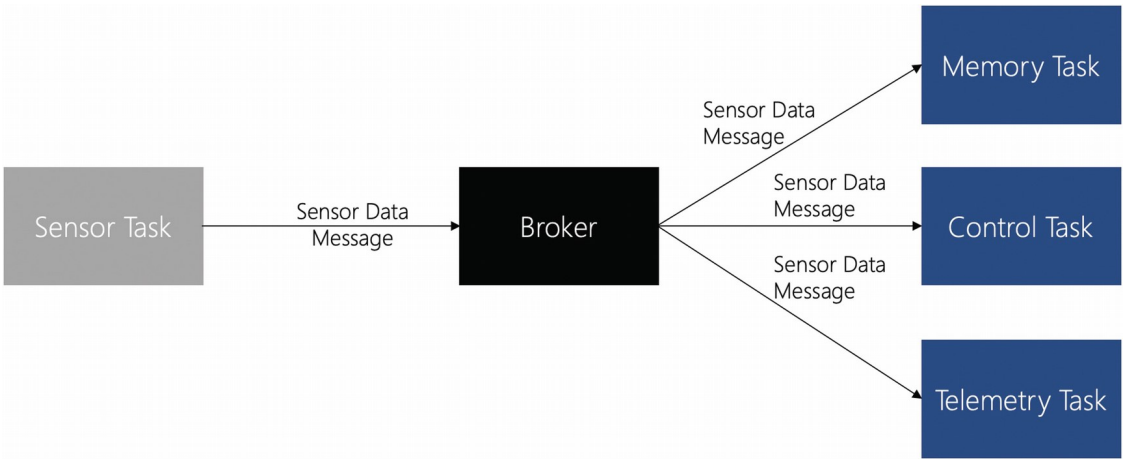

Figure 5-23 A sensor task delivers sensor data to subscribers through the broker

What's interesting about this approach is that the dependencies are broken between the sensor tasks and any tasks that need to use the data. The

system is also entirely scalable. If we suddenly need to use the sensor data for fault handling, we don't have to go back and rearchitect our system or add another dependency. Instead, we create a fault task that subscribes to the sensor data. That task will get the data it needs to operate and detect faults.

While the publish and subscribe model can be interesting, it obviously will result in a larger memory footprint for the application. However, that doesn't mean that parts of the system can't use this design pattern.

Low-Power Application Design Patterns

Designing a device today for low power has become a familiar and essential design parameter to consider. With the IoT, there are a lot of battery-operated devices that are out there. Designers want to maximize how long the battery lasts to minimize the inconvenience of changing batteries and having battery waste. There are also a lot of IoT devices that are continuously powered that should also be optimized for energy consumption. For example, a connected home could have upward of 20 connected light switches! Should all those light switches be continuously drawing 3 Watts? Probably not. That's a big waste of energy and nonrenewable energy sources.

Regarding low-power design patterns, the primary pattern is to keep the device turned off as much as possible. The software architecture needs to be event driven. Those events could be a button press, a period that has elapsed, or another trigger. In between those events, when no practical work is going to be done, the microcontroller should be placed into an appropriate low-power state, and any non-essential electronics should be turned off. A simple state diagram for the design pattern can be seen in Figure 5-24.

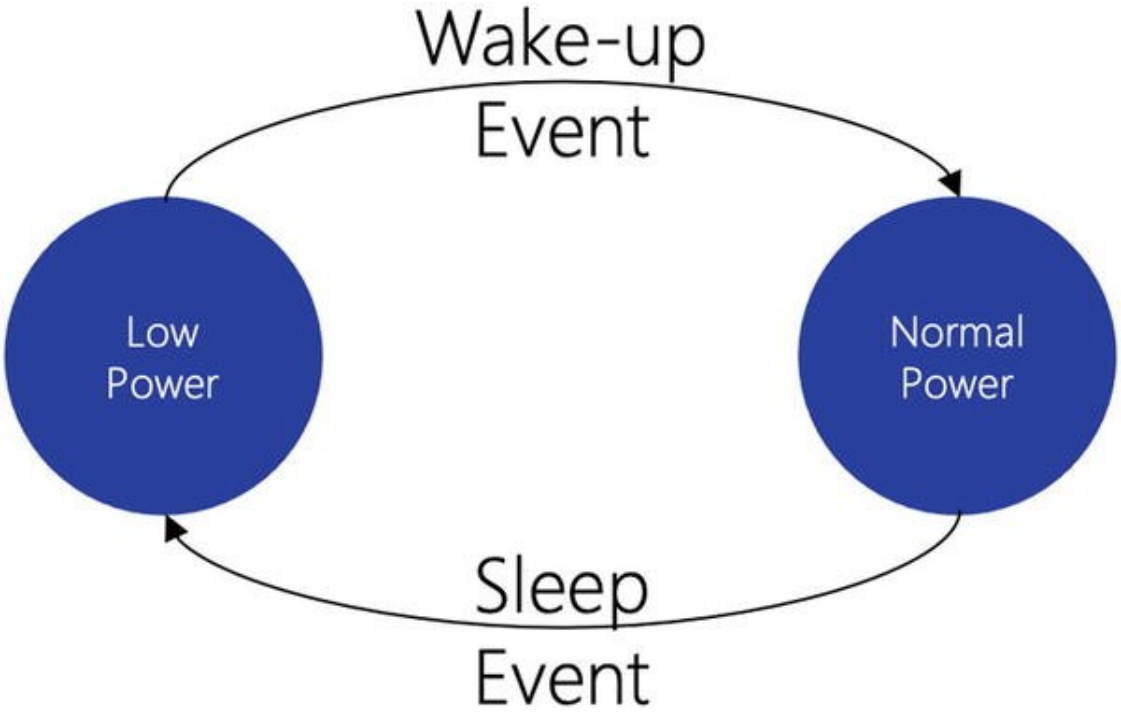

Figure 5-24 The system lives in the low-power state unless a wake-up event occurs. The system returns to a low-power state after practical work is complete

The preceding design pattern is, well, obvious. The devil is in the details of how you stay in the low-power state, wake up fast enough, and then get back to sleep. Like many things in embedded systems, the specifics are device and application specific. However, there are at least a few tips and suggestions that I can give to help you minimize the energy that your devices consume.

First, if your system uses an RTOS, one of the biggest struggles you'll encounter is keeping the RTOS kernel asleep. Most RTOS application implementations I've designed and the ones I've seen set the RTOS timer tick to one millisecond. This is too fast when you want to go into a low-power state. The kernel will wake the system every millisecond when the timer tick expires! I'd be surprised if the microcontroller and system had even settled into a low-power state by that time. Designers need to use an RTOS with a tickless mode built-in or one where they can scale their system tick to keep the microcontroller asleep for longer periods.

An example is FreeRTOS. FreeRTOS does have a tickless mode that can be enabled. The developer can add their custom code to manage the low-power state. When the system goes to sleep, the system tick is automatically scaled, so fewer system ticks occur! Figure 5-25 shows some measurements from an application I

was optimizing for energy. On channel 00, the system tick is running once every millisecond. On channel 01, task 1, there is a blip of activity and then a hundred milliseconds of no activity. The system was not going to sleep because the system tick was keeping it awake, drawing nearly 32 milliamps.

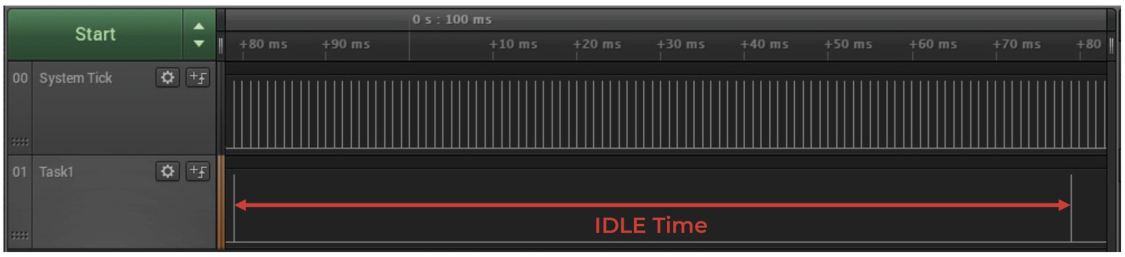

Figure 5-25 An unoptimized system whose RTOS system tick prevents the system from going to sleep

In this system, I enabled the FreeRTOS tickless mode and set up the low-power state I wanted the system to move into. I configured the tickless mode such that I could get a system tick once every 50 milliseconds, as shown in Figure 5-26. The result was that the system could move into a low-power state and stay there for much longer. The power consumption also dropped from 32 milliamps down to only 11 milliamps!

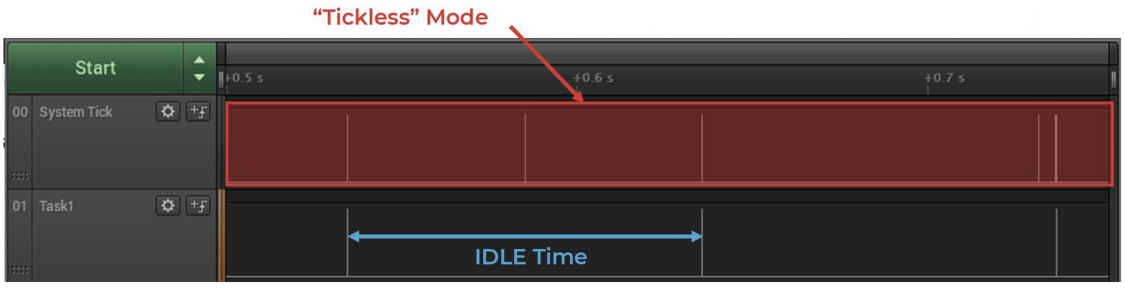

Figure 5-26 Enabling a tickless mode removed the constant system wake-up caused by the system tick

Another exciting design pattern to leverage is an event-driven architecture. Many developers don't realize it, but there is no requirement that an RTOS or an embedded system has a constantly ticking timer! You don't need a system tick! If an event causes everything that happens in the system, then there is no reason even to use the RTOS time base. The time base can be disabled. You can use interrupts to trigger tasks that synchronize with each other and eventually put the system to sleep.

Leveraging Multicore Microcontrollers

Over the last couple of years, as the IoT has taken off and machine learn-

ing at the edge has established itself, an interest in multicore microcontrollers has blossomed. Microcontrollers have always been the real-time hot rods of microprocessing. They fill a niche where hard deadlines and consistently meeting deadlines are required. However, with the newer needs and demands being placed on them, many manufacturers realized that either you move up to a general-purpose processor and lose the real-time performance, or you start to put multiple microcontrollers on a single die. Multiple processors in a single package can allow a low-power processing to acquire data and then offload it for processing on a much more powerful processor.

While multicore microcontrollers might seem new and trendy, there have been microcontrollers going back at least 20 years that have had secondary cores. In many cases, though, these secondary cores were threaded cores, where the central processor could kick off some code to be executed on the second core, and the second core would plug away at it, uninterrupted until it completed its task. A few examples are the Freescale S12X and the Texas Instruments C2000 parts. In any event, as the processing needs of the modern system have grown, so has the need for processing power. So in the microcontroller space, the current trend is to add more cores rather than making them faster and more general to compute.

Today, in 2022, there are many multicore microcontroller offerings, such as the Cypress PSoC 6, the STMicroelectronics STM32H7, and the Raspberry Pi Pico, to name a few. A multicore microcontroller is interesting because it provides multiple execution environments that can be hardware isolated where code can run in parallel. I often joke, although prophetically, that one day microcontrollers may look something like Figure 5-27, where they have six, eight, or more cores all crunching together in parallel.

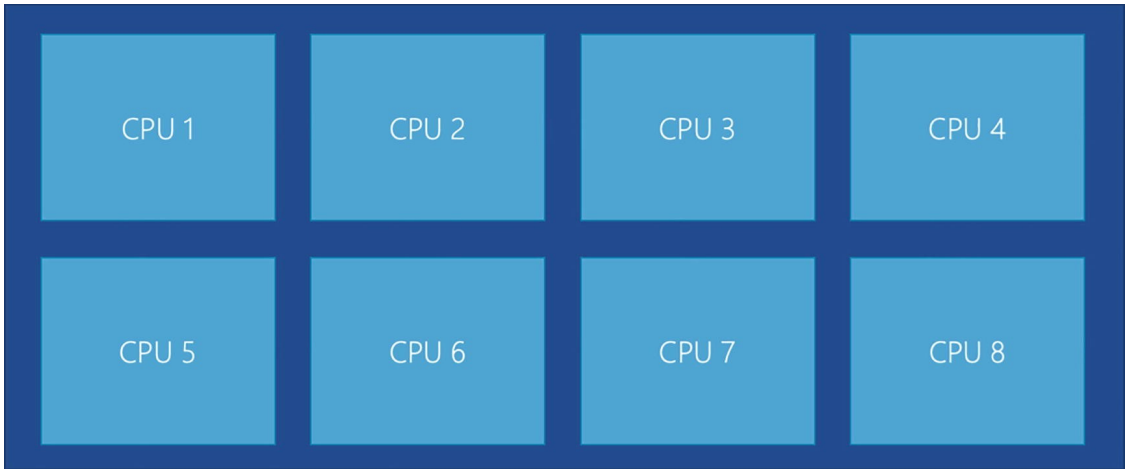

Figure 5-27 A future microcontroller is envisioned to have eight separate cores

Today's architectures are not quite this radical yet. A typical multicore microcontroller has two cores. However, two types of architectures are common. First, there is homogenous multiprocessing. In these architectures, each processing core uses the same processor architecture. For example, the ESP32 has Dual Xtensa 32-bit LX6 cores, as shown in Figure 5-28. One core is dedicated to Wi-Fi/Bluetooth, while the other is dedicated to the user application.

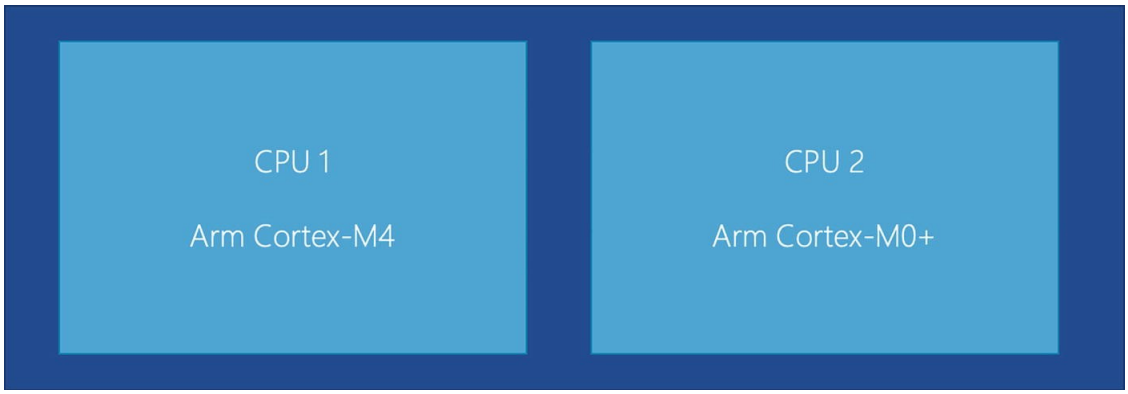

Figure 5-28 Symmetric multicore processing has two cores of the same processor architecture

The alternative architecture is to use heterogeneous multiprocessing. In these architectures, each processing core has a different underlying architecture. For example, the Cypress PSoC 64 has an Arm Cortex-M4 for user applications and an Arm Cortex-M0+ to act as a security processor (see Figure 5-29). The two cores also don't have to run at the same clock speed. For example, in the previous example, the Arm Cortex-M4 runs at 150 MHz, while the Arm Cortex-M0+ runs at 100 MHz.

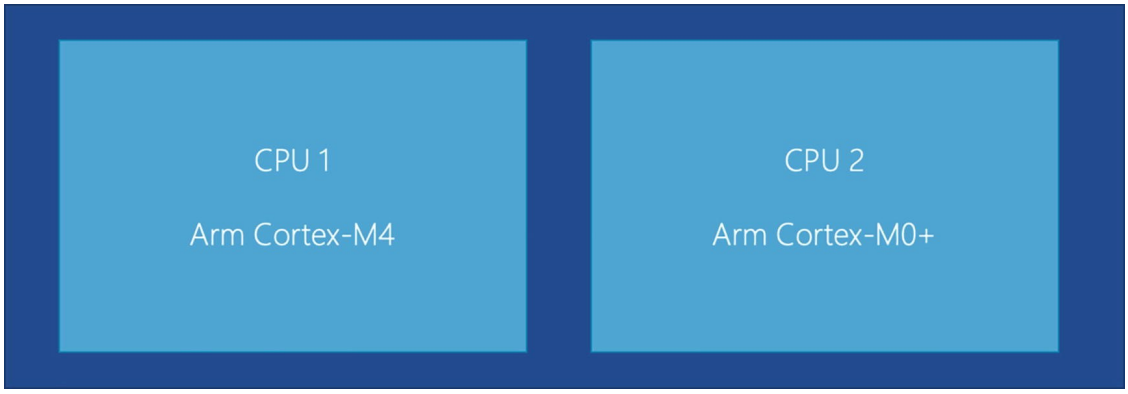

Figure 5-29 Heterogeneous multiprocessors use multiple-core architectures

With multiple processing cores, designers can leverage various use cases and design patterns to get the most effective behavior out of their embedded system. Let's explore what a few of them are.

AI and Real-Time Control

Artificial intelligence and real-time control are the first use case that I see being pushed relatively frequently. Running a machine learning inference on a microcontroller can undoubtedly be done, but it is compute cycle intensive. The idea is that one core is used to run the machine learning inference, while the other does the real-time control, as shown in Figure 5-30. For example, in the STM32H7 family, the multicore parts have an Arm Cortex-M7 and an Arm Cortex-M4. The M7 runs the machine learning inference, while the M4 does the standard real-time stuff like motor control, sensor acquisition, and communication. The M4 core can feed the M7 with the data it needs to run the AI algorithm, and the M7 can feed the M4 the result.

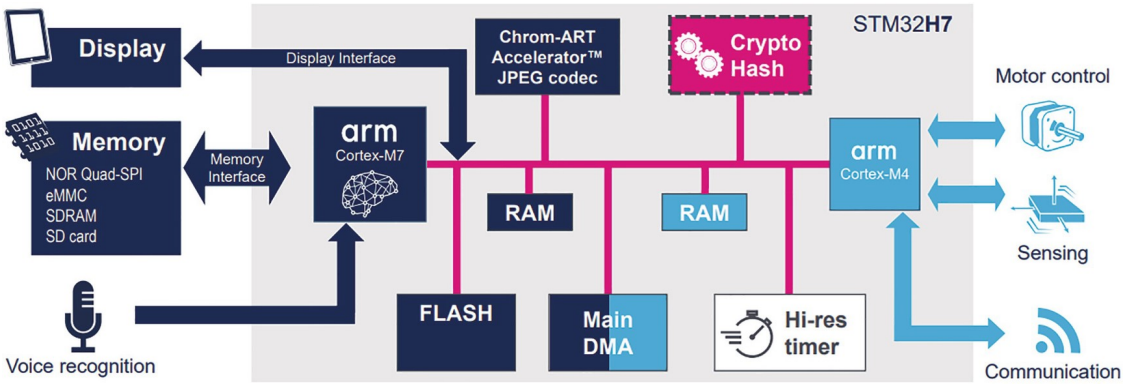

Figure 5-30 A multicore microcontroller use case where one core is used for running an AI inference, while the other core manages real-time application control (Source: STM32H7 MCUs for rich and complex applications, Slide 16)

Real-Time Control

Another use case often employed in multicore microcontroller systems is to have each core manage real-time control capabilities, as shown in Figure 5-31. For example, I recently worked on a ventilator system that used multiple cores on an STM32H7. The M7 core was used to drive an LCD and manage touch inputs. The M4 core was used to run various algorithms necessary to control pumps and valves. In addition, the M4 interfaced with a third processor that performed all the sensor data acquisition and filtering. Splitting different domain functions into different cores and processors dramatically simplified the software architecture design. It also helped with assigning responsibility to different team members and allowed for parallel development among team members.

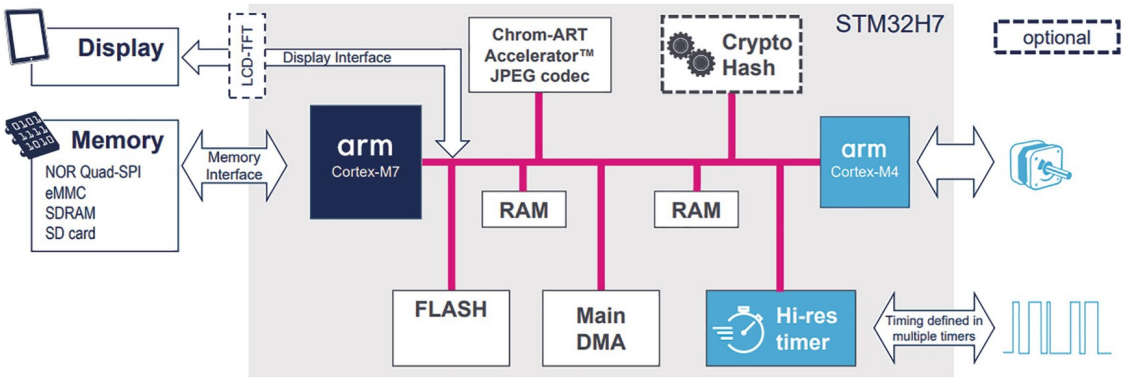

Figure 5-31 A multicore microcontroller use case where one core runs cycle-intensive real-time displays and memory, while the other manages real-time application control (Source: STM32H7 MCUs for rich and complex applications, Slide 29)

Security Solutions

Another popular use case for multiple cores is managing security solutions with an application. For example, developers can use the hardware isolation built into the multiple cores to use one as a security processor where the security operations and the Root-of-Trust live. In contrast, the other core is the normal application space (see Figure 5-32). Data can be shared between the cores using shared memory, but the cores only interact through interprocessor communication (IPC) requests.29

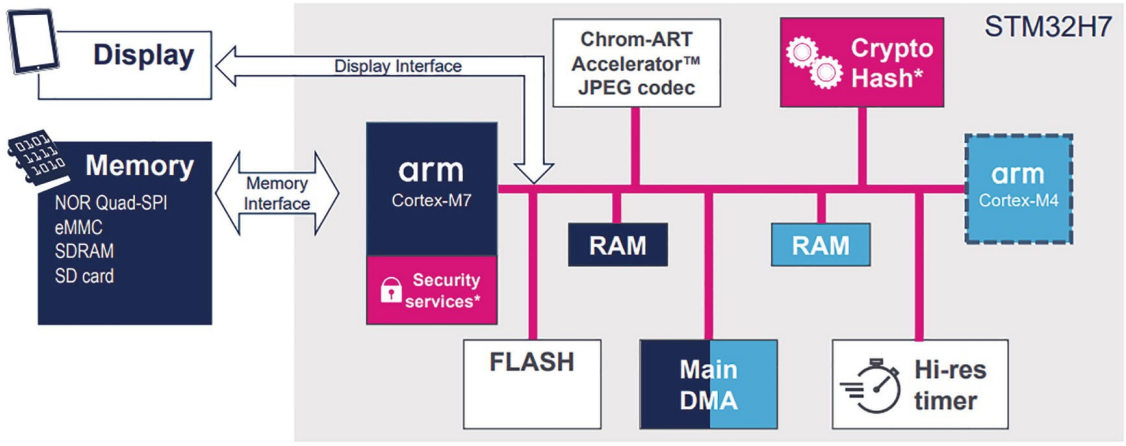

Figure 5-32 A multicore microcontroller use case where one core is used as a security processor, while the other core manages real-time application control (Source: STM32H7 MCUs for rich and complex applications, Slide 28)

A Plethora of Use Cases

The three use cases that I described earlier are just several design patterns that emerge for multicore microcontrollers. There are undoubtedly many other potential design patterns. There will certainly be others once microcontrollers start to move beyond having just two cores. Multiple microcontrollers will be an area to keep an eye on in the future. While many applications currently are targeting the high end of the spectrum, as costs come down, we will undoubtedly start seeing 8-bit and 16-bit applications for multicore parts.

Final Thoughts

Design patterns are commonly occurring patterns in software that designers and developers can leverage to accelerate their design work. There often isn't a need to design a system from scratch. Instead, the architecture can be built from the ground up using fundamental patterns. Not only does this accelerate design, but it also can help with code portability.

Action Items
To put this chapter's concepts into action, here are a few activities the reader can perform to get more familiar with design patterns:

- Examine how your most recent application accesses peripherals. For example, are they accessed using polling, interrupts, or DMA?

  - What improvements could be made to your peripheral interactions?
- Consider a product or hobby project you would be interested in building. What design patterns that we've discussed would be necessary to build the system?
- What parts, if any, of your application could benefit from using publish and subscribe patterns?
- What are a couple of design patterns that can be used to minimize energy consumption? Are you currently using these patterns in your architecture? Why or why not?
- What are some of the advantages of using a multicore microcontroller? Are there disadvantages? If so, what are they?

Footnotes

1 https://springframework.guru/gang-of-four-design-patterns/

2 The definition for reasonable amount of time is obviously open to debate and very application dependent.

3 *Real-Time Concepts for Embedded Systems* by Qing Li with Caroline Yao, page 231.

4 *Real-Time Concepts for Embedded Systems* by Qing Li with Caroline Yao, page 233.

5 Modified diagram from *Real-Time Concepts for Embedded Systems* by Qing Li with Caroline Yao, page 232, Figure 15.1.

6 I teach a course entitled Designing and Building RTOS-based Applications that goes further into these patterns.