J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_2

2. Embedded Software Architecture Design

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

If I could describe my first few years as an embedded software engineer, I would use the words “code first, architect second.” I could simplify this statement further to “There is only code!”. Creating a software architecture was honestly a foreign concept, and it was an industry-wide issue, not just for me. When a task needed to be done, we would jump into our IDE and start coding. For simple projects, this wasn’t an issue. Everything that needed to be done could be kept straight in our minds; however, as projects became more complex, the rework required to add features ballooned out of control.

I recall the first time I went against the wisdom of the time and decided to draw out and design what I was doing before coding it up. It was like I had been coding in the dark my whole life, and suddenly the clouds cleared, the sun shone, and I understood! Architect first and code second became my mantra. Developing a software architecture allowed me to think through what I would code, adjust how it worked, and then code it once. Writing code went from constantly rewriting, modifying, and debugging to designing and implementing. The time required decreased by literally 10x! I had discovered what Ralf Speth had previously stated:

> *If you think good design is expensive, you should look at the cost of bad design.*

The IEEE 1471 standard defines a software architecture as follows:

> **Definition** *A software architecture is the fundamental organization of a system embodied in its components, their relationship to each other and the environment, and the principles* **guiding its design and evolution.**[1]

This simple statement tells us much about software architecture and why it is essential to the software development life cycle. Software architecture is a developer's road map, the GPS, on what they are supposed to be building. Without a software architecture, it is straightforward for a developer to get lost and make wrong turns, which forces them to continually retrace their steps, trying to figure out not just how to get where they want to go but where they are trying to go!

A map tells the reader the lay of the land. Landmarks, buildings, and roads provide a general organization of an area and the relationship between those objects. A software architecture does the same thing for software; instead of physical objects and spatial relationships, we are dealing with components, objects, and the relationships between them. The software architecture tells a developer what they are building; it doesn't tell them how to make it!

This brings us to an important point: a software architecture should be hardware independent! The software architecture should guide a developer to successfully write an application, whether it is written for a microcontroller or application processor. The software architecture is also agnostic to the programming language that is used to implement the architecture! For a given architecture, I can write the application in C, C++, Python, Rust, or any other programming language that may suit my fancy.[2]

The software architecture is a map, a blueprint, and a guide for the developer to follow. The developer can decide how they want to implement the architecture; they aren't told by the architecture how to do it. Software architecture can readily be compared to the blueprints used by a builder to construct a building. The blueprints tell them the general footprint, how many stories, and where the elevators and stairwells go. However, they don't tell the electrician how to wire a light switch or where the occupant should put the furniture.

> **Beware** It is a rare occasion, but sometimes the hardware architecture is so unique that it can influence the software architecture!

The software architecture also is not set in stone! Instead, it provides

principles that guide the design and implementation of the software. It's so important to realize that software design will evolve. It shouldn't devolve into chaos like so many software systems do but evolve to become more efficient, include additional features, and meet the needs of its stakeholders. I can't tell you a single system I've ever worked on where we got all the details up front, and nothing changed. Software design, by nature, is an exercise in managing the unknown unknowns. To manage them properly, the software architecture must quickly evolve with minimal labor.

Now you might be thinking, I don't need a software architecture. I can design my software on the fly while I'm implementing it. (A lot of open source software is developed this way). The problem, though, is that embedded software today has become so complex that it's nearly impossible to implement it efficiently and timely without an architecture. For example, if you were to design and erect a building, would you just buy concrete and lumber and start building without blueprints? Of course, not! That would be a foolhardy endeavor. Yet, many software products are just as complex as a building, and developers try to construct them without any plan!

Developing a software architecture provides many benefits to developers. Just a few highlights include

- A plan and road map to what is being built
- A software picture that can be used to train engineers and explain the software to management and stakeholders
- Minimizing rework by minimizing code changes
- Decreased development costs
- A clear picture of what is being built
- An evolvable code base that can stand the tests of time (at least product lifetimes)

Now that we understand what software architecture is and what it can do for us, let's discuss an interesting architectural challenge that embedded software developers face.

A Tale of Two Architectures

Embedded software designs are different when compared to web, mobile, or PC software architectures. General applications have just a single architecture that is required, a hardware-independent business rules–based architecture. These applications are written to be general and can run across multiple hardware architectures and have zero hardware dependence. The architecture focuses on the business logic and the high-level components required to get the job done.

Embedded software designs have a second type of architecture that includes a hardware-dependent real-time architecture. The hardware-dependent architecture requires special knowledge of the hardware the application will be running on. It allows the designer to optimize their architecture for different hardware platforms, if necessary, or just optimize it to run most efficiently on a single architecture. For example, a designer may need to understand the complex behavior of available watchdogs, memory protection units (MPUs), direct memory access (DMA), graphics accelerators, communications buses, etc. This architecture is where the "rubber meets the road" or, in our case, where the application architecture meets real-time hardware.

> **Best Practice***Embedded software architects should separate their architectures into two pieces: business logic and real-time hardware-dependent logic.*

I can't stress enough how important it is to recognize that these two different architectures exist in an embedded system. If designers overlook this, they risk developing an architecture that tightly couples the business logic to the hardware. While this may not seem like a problem at first, in today's systems, this can lead to code that is difficult to maintain, port, and scale to meet customer needs. The software architecture should maintain as much independence from the hardware due to the potential for disruptions to occur in the supply chain. If a microcontroller becomes unavailable or obsolete, or suddenly a dramatic increase in lead time occurs, it will be necessary to pivot to other hardware. Software that is tightly coupled to the hardware will make the change expensive and

time-consuming.

The separation between the application business architecture and the real-time software architecture can be seen from a 30,000-foot view in Figure 2-1. Note that the two architectures are separated and interact through an abstraction layer.

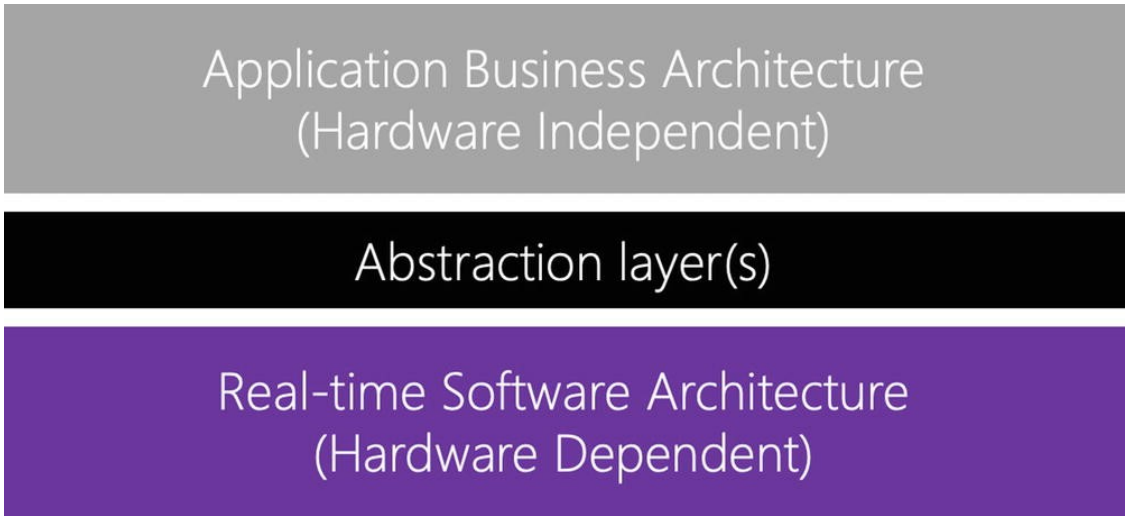

Figure 2-1 Embedded software architectures are broken into two architectures: a hardware-independent business layer and a hardware-dependent real-time layer

Definition*An abstraction layer is software that translates a high-level request into the low-level commands required to operate.*[3]

Approaches to Architecture Design

Generally, two approaches are typically used to develop an embedded software architecture: the bottom-up approach and the top-down approach. The general difference between these approaches can be seen in Figure 2-2, but let's discuss each and see how they impact the modern designer.

Most embedded software developers I know tend to be naturally inclined to use the bottom-up approach. This approach starts at the hardware and driver layer to identify the software components and behaviors. Once the low-level pieces are placed, the designer works their way up the stack into the application code. The core idea is that you build up the foundation of the software system first and then work your way up to the product-specific architecture. Figure 2-2 demonstrates how this works with our two different software architectures.

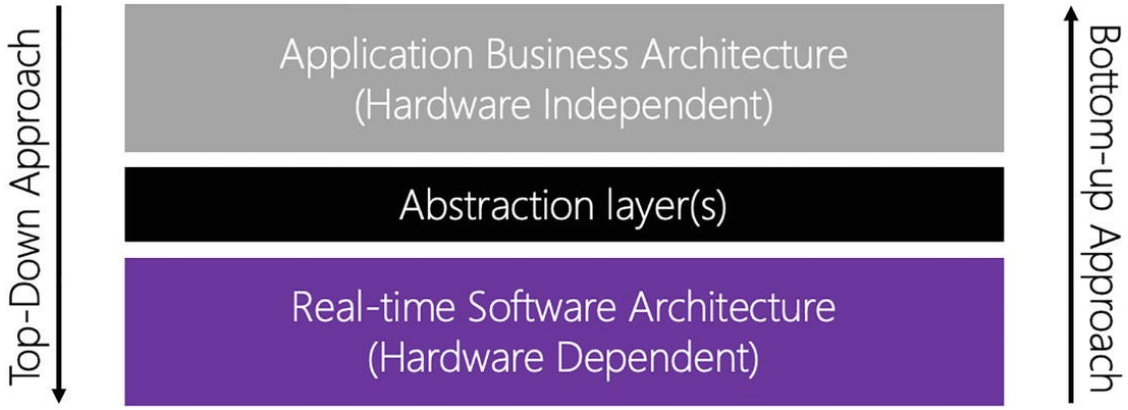

Figure 2-2 Embedded software architectures are developed using two approaches: top-down and bottom-up

The top-down approach tends to be more comfortable and the starting point for embedded software developers who have more of an application background than a hardware background. The top-down approach starts with the high-level application and the business logic and then works its way down to the abstraction layers and the hardware. The core idea here is that the product differentiator is not in the foundation but the business logic of the device! These designers want to focus on the application code since this is where product differentiation is achieved. The hardware and lower layers are just a means to the end, and it doesn't matter to the product features or the stakeholders (unless it's built upon a poor framework or slow processor).

Beware*The goal is to produce a product that generates a profit! Everything else is tertiary (or less).*

With two approaches available to us, which one is the right one? Well, it depends on your team and the resources that they have available to them. However, I believe that the right approach for most teams today is to use the top-down approach. There are several reasons why I think this.

First, designers will focus more on their product features and differentiators, the secret sauce that will help sell their products. The more focused I am on my product features, the more I can think through, prioritize, and ensure that my design is going in the right direction. If I save it for the end, I will likely "mess up" on my features and provide a poor experience to the product users.

Second, modern development tools, which we will discuss later in the book, can be used to model and simulate the application's behavior long before the hardware can. This allows us to put the application code to the test early. The earlier we can iterate on our design and adjust before coding on the target, the less expensive development will be. The change cost grows exponentially as a project progresses through the various development stages, as seen in Figure 2-3.[4] The sooner we can catch issues and mistakes, the better it will be from a cost and time-to-market standpoint.

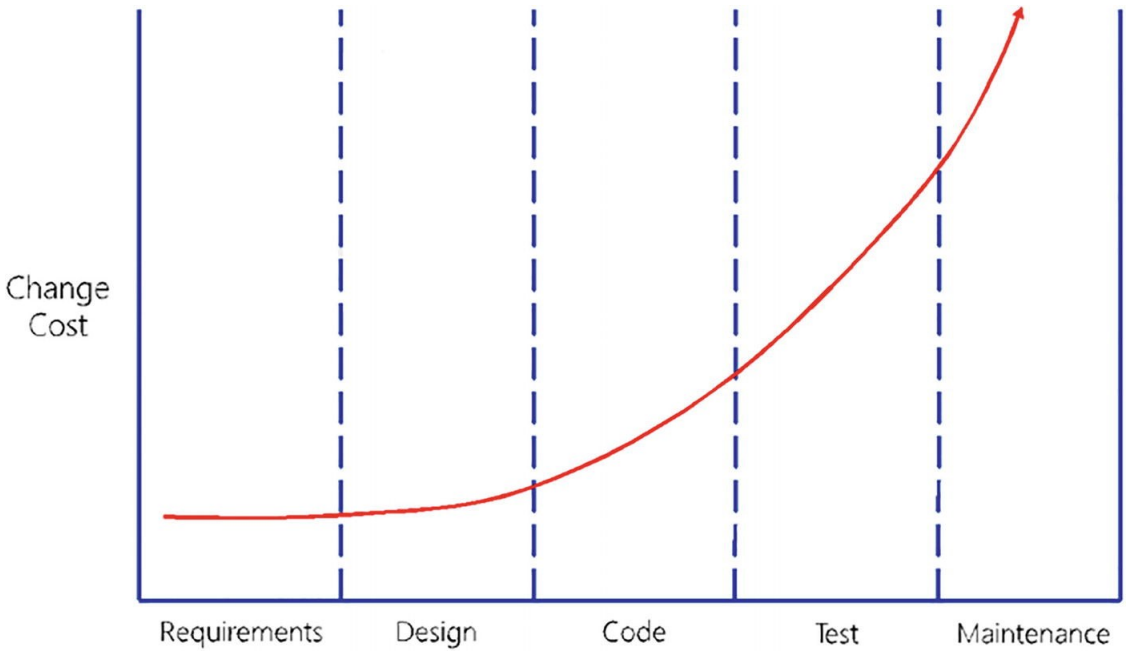

Figure 2-3 The change cost grows exponentially as a project progresses through the various development stages

Finally, by designing the top level first, simulating it, and getting feedback, if things don't go well, we can fail quickly. In business, nothing is worse than investing in something that drags on, consumes the budget, and then fails. Companies want projects that are going to fail to fail quickly.[5] If a product isn't going to work, the sooner we know, the better, and the faster we can move on to the next idea. Starting at the top can help us validate our application with the target audience sooner.

There is a hybrid approach that I sometimes take with my customers that have a large enough team and resources to make it work. It's where we simultaneously design both architectures! Some team members are assigned to focus on the high-level business logic architecture, while others are given to the low-level real-time architecture. Honestly, this can be a

great approach! We essentially have two different application domains that need to interact through a well-defined abstraction layer. There is no reason why these activities cannot be parallelized and meet in the middle!

Characteristics of a Good Architecture

One concern many developers and software architects have is whether the architecture is good; in other words, does it meet their needs. As strange as it sounds, I see many teams just jump in and start to design their architecture without defining what their end goals are! Of course, every system won't have the same goals; however, some common characteristics that most embedded software architects aim for help us define whether the architecture is good or not.

Many teams aim to ensure that their architecture is reusable, scalable, portable, maintainable, and so forth. Early in the architectural phase, developers should write down their goals for the architecture. If a team is developing a quick rapid prototype, who cares if the architecture is portable or maintainable? However, if the code is the base for multiple products, scalable and maintainable goals are good targets.

No matter what the end goals are for the software, there are two characteristics that every architect needs to carefully manage to reach their end goals: coupling and cohesion.

Architectural Coupling

Coupling refers to how closely related different modules or classes are to each other and the degree to which they are interdependent.[6] The degree to which the architecture is coupled determines how well a developer can achieve their architectural goals. For example, if I want to develop a portable architecture, I need to ensure that my architecture has low coupling.

There are several different types and causes for coupling to occur in a software system. First, common coupling occurs when multiple modules

have access to the same global variable(s). In this instance, code can't be easily ported to another system without bringing the global variables along. In addition, the global variables become dangerous because they can be accessed by any module in the system. Easy access encourages "quick and dirty" access to the variables from other modules which then increases the coupling even further. The modules have a dependency on those globally shared variables.

Another type of coupling that often occurs is content coupling. Content coupling is when one module accesses another module's functions and APIs. While at first this seems reasonable because data might be encapsulated, developers have to be careful how many function calls the module depends on. It's possible to create not just tightly coupled dependencies but also circular dependencies that can turn the software architecture into a big ball of mud.

Coupling is most easily seen when you try to port a feature from one code base to another. I think we've all gone through the process of grabbing a module, dropping it into our new code base, compiling it, and then discovering a ton of compilation errors. Upon closer examination, there is a module dependency that was overlooked. So, we grab that dependency, put it in the code, and recompile. More compilation errors! Adding the new module quadrupled the number of errors! It made things worse, not better. Weeks later, we finally decided it's faster to just start from scratch.

Software architects must carefully manage their coupling to ensure they can successfully meet their architecture goals. Highly coupled code is always a nightmare to maintain and scale. I would not want to attempt to port highly coupled code, either. Porting tightly coupled code is time-consuming, stressful, and not fun!

Architectural Cohesion

The coupling is only the first part of the story. Low module coupling doesn't guarantee that the architecture will exhibit good characteristics and meet our goals. Architects ultimately want to have low coupling and high cohesion. Cohesion refers to the degree to which the module or class

elements belong together.

In a microcontroller environment, a low cohesion example would be lumping every microcontroller peripheral function into a single module or class. The module would be significant and unwieldy. Instead, a base class could be created that defines the common interface for interacting with peripherals. Each peripheral could then inherit from that interface and implement the peripheral-specific functionality. The result is a highly cohesive architecture, low coupling, and other desirable characteristics like reusable, portable, scalable, and so forth.

Cohesion is really all about putting “things” together that belong together. Code that is highly cohesive is easy to follow because everything needed is in one place. Developers don’t have to search and hunt through the code base to find related code. For example, I often see developers using an RTOS (real-time operating system) spread their task creation code throughout the application. The result is low cohesion. Instead, I pull all my task creation code into a single module so that I only have one place to go. The task creation code is highly cohesive, and it’s easily ported and configured as well. We’ll look at an example later in the Development and Coding Skills part of the book.

Now that we have a fundamental understanding of the characteristics we are interested in as embedded architects and developers, let’s examine architectural design patterns that are common in the industry.

Architectural Design Patterns in Embedded Software

Over the years, several types of architectural patterns have found their way into embedded software. It’s not uncommon for systems to have several architectural designs depending on the system and its needs. While there are many types of patterns in software engineering, the most common and exciting for microcontroller-based systems include unstructured monoliths, layered monoliths, event-driven architectures, and microservices. Let’s examine each of these in detail and understand the pros and

cons of each.

The Unstructured Monolithic Architecture

The unstructured monolithic software architecture is the bane of a modern developer's existence. Sadly, the unstructured monolith is easy to construct but challenging to maintain scale and port. In fact, given the opportunity, most code bases will try their best to decay into an unstructured monolith if architects and developers don't pay close attention.

The great sage Wikipedia describes a monolithic application as "a single-tiered software application in which the user interface and data access code are combined into a single program from a single platform."[7] An unstructured monolithic architecture might look something like Figure 2-4. The architecture is tightly coupled which makes it extremely difficult to reuse, port, and maintain. Individual modules or classes might have high cohesion, but the coupling is out of control.

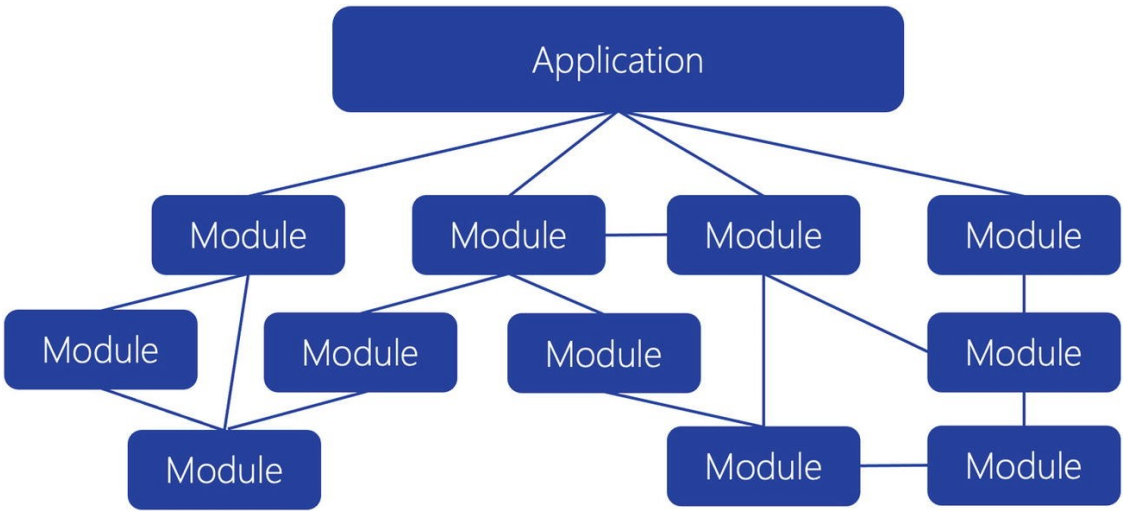

Figure 2-4 Unstructured monolithic architectures are tightly coupled applications that exist in the application layer

An unstructured monolith was one of the most common architectures in embedded systems when I started back in the late 1990s. A real-time microcontroller-based system was viewed as a "hot rod"; it was designed for a single, one-off purpose, and it needed to be fast and deterministic. For that reason, systems often weren't designed with reuse in mind. Fast forward 20–30 years, microcontrollers are performance powerhouses with lots of memory. Systems are complex, and building that hot rod from

scratch in an unstructured monolith doesn't meet today's needs.

Layered Monolithic Architectures

Although I have no data to prove, I would like to argue that layered monolithic architectures are the most common architecture used in embedded applications today. The layered architecture allows the architect to separate the application into various independent layers and only interact through a well-defined abstraction layer. You may have seen layered monolithic architectures like Figure 2-5.

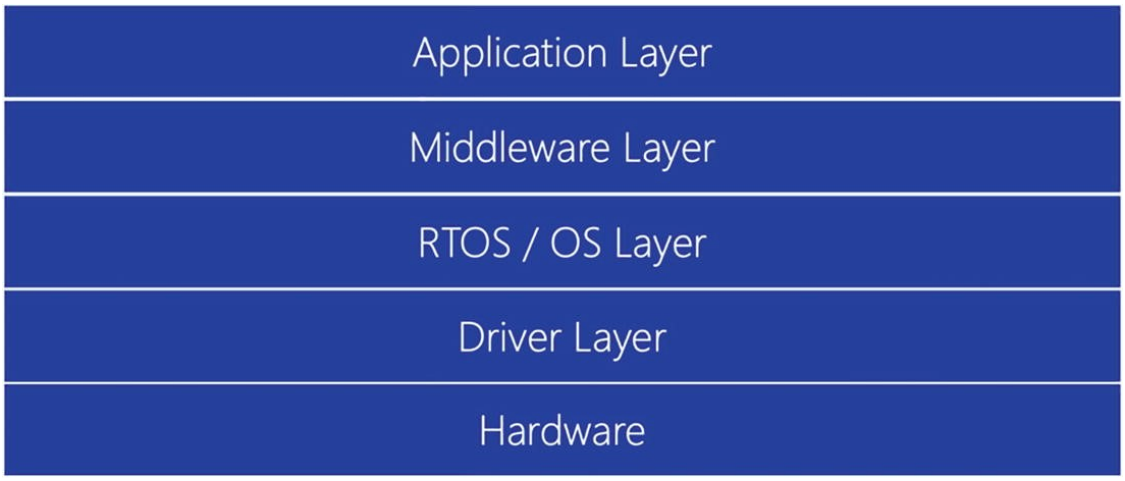

Figure 2-5 An example is layered monolithic architecture for an embedded system

A layered monolithic application attempts to improve the high coupling of an unstructured monolithic architecture by breaking the application into independent layers. Each layer should only be allowed to communicate with the layer directly above or below it through an abstraction layer. The layers help to break the coupling, which allows layers to be swapped in and out as needed.

The most common example of swapping out a layer is when the architecture will support multiple hardware platforms. For example, I've worked with clients who wanted to use Microchip parts in one product, NXP parts in another, and STM parts in another. The products were all related and needed to run some standard application components. Instead of writing each product application from scratch, we designed the architecture so that the application resembled Figure 2-5. The drivers were placed behind a standard hardware abstraction layer (HAL) that made the application dependent on the HAL, not the underlying hardware.

Leveraging APIs and abstraction layers is one of the most significant benefits of layered monolithic architecture. It breaks the coupling between layers, allows easier portability and reuse, and can improve maintainability. However, one of the biggest gripes one could have with the layered monolithic architecture is that they break the “hot rodness” of the system.

As layers are added to an application, the performance can take a hit since the application can no longer just access hardware directly. Furthermore, clock cycles need to be consumed to circumnavigate the layered architecture. In addition to the performance hit, it also requires more time up front to design the architecture properly. Finally, the code bases do tend to be a little bit bigger. Despite some disadvantages, modern microcontrollers have more than enough processing power in most instances to overcome them.

DefinitionAn OSAL is an operating system abstraction layer. An OSAL can decouple the application dependency on any one specific operating system.

Before we look at the next architecture type, I’d like to point out two critical layered architectures. First, notice that each layer can still be considered an unstructured monolithic architecture. Architects and developers, therefore, need to be careful in each layer that they manage coupling and cohesion carefully. Second, each layer in the layered architecture does not have to extend across the entire previous layer. Like a printed circuit board (PCB) layer, you can create islands and bypasses to improve performance. An example modern layered architecture diagram can be seen in Figure 2-6.

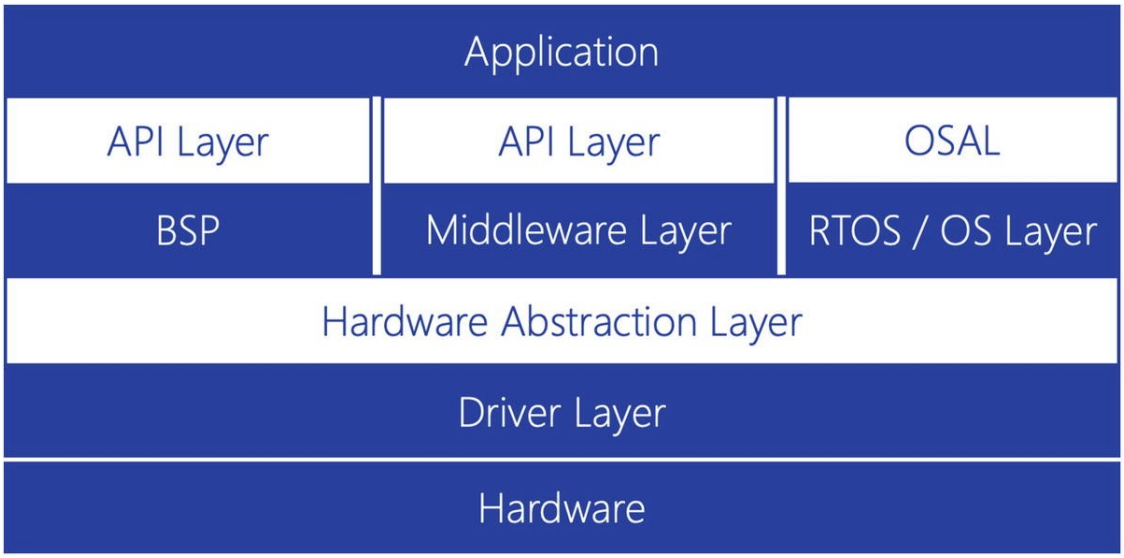

Figure 2-6 An example is modern layered monolithic architecture for an embedded system

Figure 2-6 has many benefits. First, note how we can exchange the driver layer for working with nearly any hardware by using a hardware abstraction layer. For example, a common HAL today can be found in Arm's CMSIS.[8] The HAL again decouples the hardware drivers from the above code, breaking the dependencies.

Next, notice how we don't even allow the application code to depend on an RTOS or OS. Instead, we use an operating system abstraction layer (OSAL). If the team needs to change RTOSes, which does happen, they can just integrate the new RTOS without having to change a bunch of application code. I've encountered many teams that directly make calls to their RTOS APIs, only later to decide they need to change RTOSes. What a headache that creates! An example of OSAL can be found in CMSIS-RTOS2.[9]

Next, the board support package exists outside the driver layer! At first, this may seem counterintuitive. Shouldn't hardware like sensors, displays, and so forth be in the driver layer? I view the hardware and driver layer as dedicated to only the microcontroller. Any sensors and so on that are connected to the microcontroller should be communicated through the HAL. For example, a sensor might be on the I2C bus. The sensor would depend on the I2C HAL, not the low-level hardware. The abstraction dependency makes it easier for the BSP to be ported to other applications.

Finally, we can see that even the middleware should be wrapped in an abstraction layer. If someone is using a TLS library or an SD card library, you don't want your application to be dependent on these. Again, I look at this as a way to make code more portable, but it also isolates the application so that it can be simulated and tested off target.

Suppose you want to go into more detail about this type of architecture and learn how to design your hardware abstraction layer and drivers. In that case, I'd recommend reading my other book *Reusable Firmware Development*.

Event-Driven Architectures

Event-driven architectures make a lot of sense for real-time embedded applications and applications concerned with energy consumption. In an event-driven architecture, the system is generally in an idle state or low-power state unless an event triggers an action to be performed. For example, a widget may be in a low-power idle state until a button is clicked. Clicking the button triggers an event that sends a message to a message processor, which then wakes up the system. Figure 2-7 shows what this might look like architecturally.

Figure 2-7 An example modern layered monolithic architecture for an embedded system

Event-driven architectures typically utilize interrupts to respond to the event immediately. However, processing the event is usually offloaded to a central message processor or a task that handles the event. Therefore, event-driven architectures often use message queues, semaphores, and event flags to signal that an event has occurred in the system. Using an RTOS for an event-driven architecture is helpful but not required. (We will discuss more application design using an RTOS in Chapter 4.)

The event-driven architecture has many benefits. First, it is relatively scalable. For example, if the widget in Figure 2-7 needed to filter sensor data when a new sample became available, the architecture could be modified to something like Figure 2-8. New events can easily be added to the software by adding an event handler, an event message, and the function that handles the event.

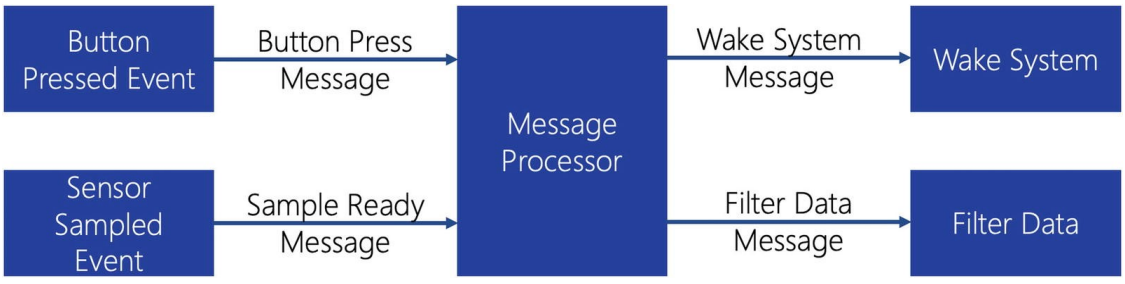

Figure 2-8 The event-driven architecture is scalable and portable. In this example, a sensor sampled event has been added

Another benefit to the event-driven architecture is that software modules

generally have high cohesion. Each event can be separated and focused on just a single purpose.

One last benefit to consider is that the architecture has low coupling. Each event minimizes dependencies. The event occurs and needs access only to the message queue that is input to the central message processor. Message queue access can be passed in during initialization, decoupling the module from the messaging system. Even the message processor can have low coupling. The message processor needs access to the list of messages it accepts and either the function to execute or a new message to send. Depending on the implementation, the coupling can seem higher. However, the message processor can take during its initialization configuration tables to minimize the coupling.

The disadvantage to using an event-driven architecture with a central message processor is that there is additional overhead and complexity whenever anything needs to be done. For example, instead of a button press just waking the system up, it needs to send a message that then needs to be processed that triggers the event. The result is extra latency, a larger code base, and complexity. However, a trade-off is made to create a more scalable and reusable architecture.

If performance is of concern, architects can use an event-driven architecture that doesn't use or limits the use of the central message processor. For example, the button press could directly wake the system, while other events like a completed sensor sample could be routed through the message processor. Such an architectural example can be seen in Figure 2-9.

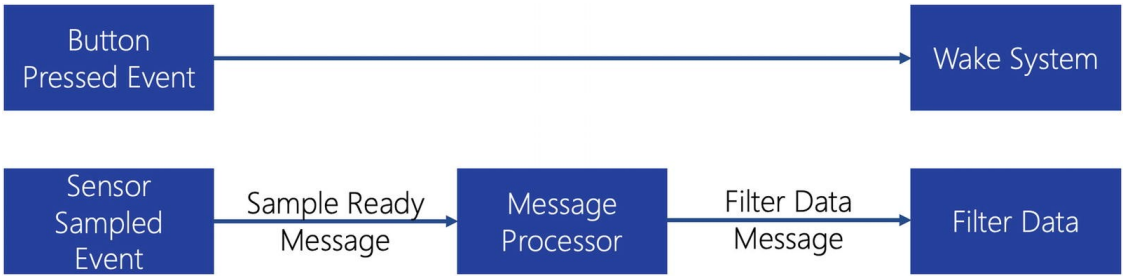

Figure 2-9 To improve performance, the event-driven architecture allows the button-pressed event to circumvent the message processor and directly perform the desired action

It's not uncommon for an architectural solution to be tiered. A tiered ar-

chitecture provides multiple solutions depending on the real-time performance constraints and requirements placed on the system. A tiered architecture helps to balance the need for performance and architectural elegance with reuse and scalability. The tiered architecture also helps to provide several different solutions for different problem domains that exist within a single application. Unfortunately, I often see teams get stuck trying to give a single elegant solution, only to talk themselves in circles. Sometimes, the simplest solution is to tier the architecture into multiple solutions and use the solution that fits the problem for that event.

Microservice Architectures

Microservice architectures are perhaps the most exciting yet challenging architectures available to embedded architects. A microservice architecture builds applications as a collection of small autonomous services developed for a business domain.[10] Microservices embody many modern software engineering concepts and processes such as Agile, DevOps, continuous integration, and continuous deployment (CI/CD).

Microservices, by nature, have low coupling. Low coupling makes the microservice maintainable and testable. Developers can quickly scale or port the microservice. If a new microservice is developed to replace an old one, it can easily be swapped out without disrupting the system. Engineering literature often cites that microservices are independently deployable, but this may not always be true in constrained embedded systems that use a limited operating system or without a microservice orchestrator to manage the services.

A significant feature of microservices is that they are organized around the system's business logic. Business logic, sometimes referred to as business capabilities, are the business rules and use cases for how the system behaves. I like to think of it as the features that a customer is paying to have in the product. For example, a customer who buys a weather station will be interested in retrieving temperature and humidity data. They don't care the sensor that provides the temperature is a DHT20. The business logic is that a temperature request comes in, and the business layer

provides a response with the temperature data.

For embedded systems, microservices can go far beyond just the system's business logic. For example, a weather station might contain microservices for telemetry, sensor interfacing, motor control (to maximize sun illumination for battery charging), and a command service.

The sensor service would be responsible for directly interacting with sensors such as temperature, humidity, motor position, etc. The service would then report this information to the motor and telemetry services. The motor service would drive the motor based on commands from the command service. Status information from the motor service would be reported to the telemetry service. Finally, the command service could command the motor and provide status updates to telemetry. The user could interact with the system, whether connected via a browser, smart app, or whatever. The example architecture can be seen in Figure 2-10.

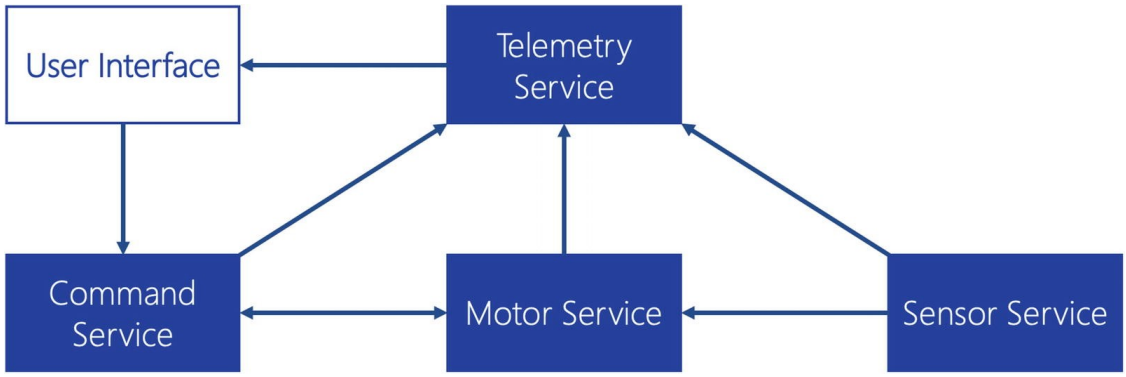

Figure 2-10 An example weather station microservice architecture. Each service is independent and uses messages to communicate with other services

At first glance, the microservice architecture may not seem very decoupled. This is because we have not yet discussed what the microservice block looks like. A microservice is made up of five pieces: the microservice, an incoming queue, outbound messages, a log, and microservice status. Figure 2-11 shows each of these pieces.

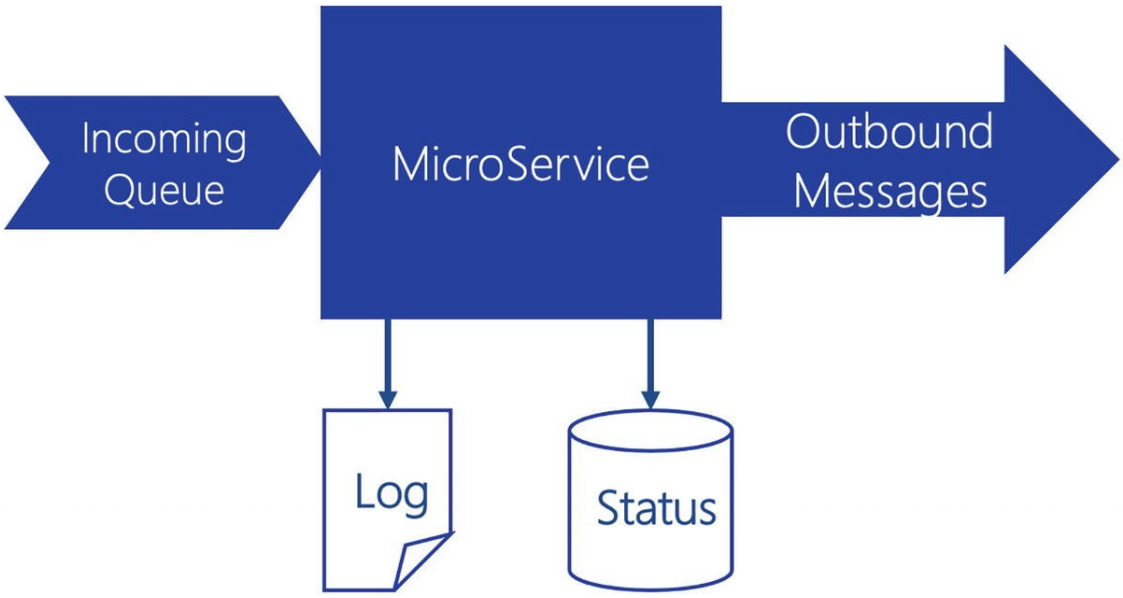

Figure 2-11 A microservice comprises five standard components, which are graphically represented here

A microservice is typically a process running on the microprocessor that contains its memory and consists of at least a single task or thread. For systems using an RTOS, a microservice can be just a single task or multiple tasks that use the memory protection unit (MPU) to create a process. The microservice contains all the code and features necessary to fulfill the service's needs.

The microservice contains a message queue used to receive messages from other microservices. In today's systems, these could be messages from other microservices running on the microprocessor or messages obtained from microservices running in the cloud or other systems! The microservice also can send messages to other services.

The important and sometimes overlooked features of a microservice by embedded developers are the logging and status features. The microservice should be able to track its behavior through a logging feature. Information that can be logged varies, but it can include messages received, transmitted, and so forth. In addition, the microservice status can be used to monitor the health and wellness of the microservice.

As we have seen, microservices have a lot of advantages. However, there are also some disadvantages. First, architecturally, they can add complexity to the design. Next, they can add extra overhead and memory needs by having communication features that may not be needed in other archi-

tectures. The decentralized nature of the architecture also means that real-time, deterministic behavior may be more challenging. Timing and responses may have additional jitter.

Despite the disadvantages, the microservice architecture can be advantageous if used correctly. However, microservice architectures may not fit your needs or product well. Furthermore, trying to do so could cause increased development times and budgets. Therefore, it's essential to carefully analyze your needs and requirements before committing to any architecture.

Before selecting the best architecture for your application, it's also essential to consider the application domains involved in the product. Architects should decompose their applications into different domains.

Application Domain Decomposition

A growing number of products require that the system architecture execute across multiple domains. For example, some code may run on an embedded controller while other pieces run in the cloud. Application domain decomposition is when the designer breaks the application into different execution domains. For experienced embedded developers, this might initially seem strange because designers working with microcontroller-based systems lump all their code into a single execution domain. They view all code as running on a single CPU, in a single domain, in one continuous memory map.

It turns out that there are four common application domains that every embedded designer needs to consider. These include

- The Privilege Domain
- The Security Domain
- The Execution Domain
- The Cloud Domain

Let's examine each of these in greater detail and understand the unique insights these domains can bring to the design process.

The Privilege Domain

Just a moment ago, I mentioned that many embedded software designers lump all their code into a single execution domain. They view all code running on a single CPU in one continuous memory map. While this may be true at the highest levels, developers working with microcontroller will often find that the processor has multiple privilege levels that the code can run in. For example, an Arm Cortex-M processor may have their code running in privileged or unprivileged modes.

In privileged mode, code that is executing has access to all CPU registers and memory locations. In unprivileged mode, code execution is restricted and may not access certain registers or memory locations. Obviously, we don't just want to have all our code running in privileged mode! If the code runs off into the weeds, it would be nice to prevent the code from accessing critical registers.

Within the Cortex-M processor, it's interesting to also note that there are also two modes that code can be running in which affect the privileged mode: thread and handler modes. Thread mode is used for normal code execution, and the code can be in either the privileged or unprivileged mode. Handler mode is specifically for executing exception handlers and only runs in privileged mode. A typical visual summary for these modes can be found in Figure 2-12.

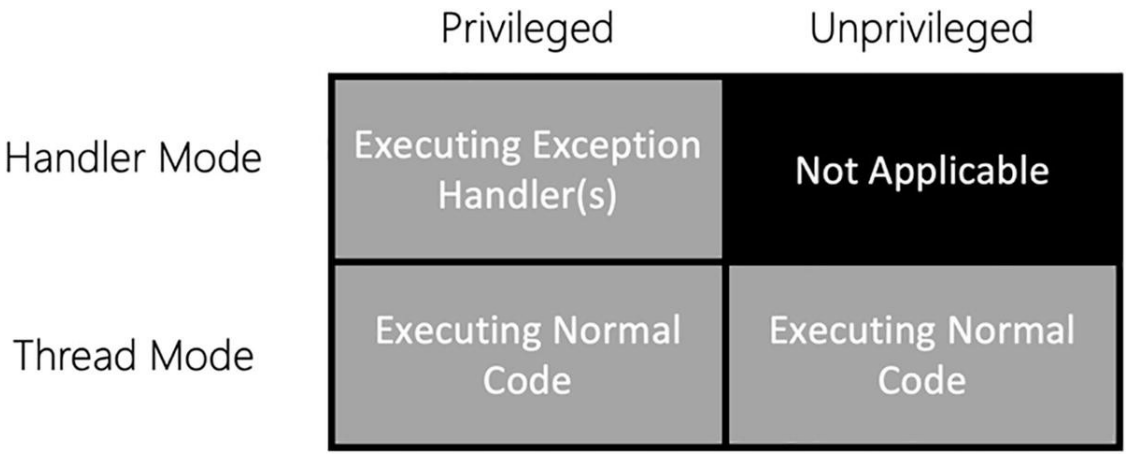

Figure 2-12 A summary of the privilege states and modes is on Arm Cortex-M microcontrollers[11]

So, while a developer may think their application is 100% in a single domain, the designer should carefully consider the processor modes in which the application should run.

The Security Domain

The security domain is an important domain for designers to consider, although unfortunately one that is often overlooked by developers. Standalone systems require security to prevent tampering and intellectual property theft as much as a connected device does. However, the attack surface for a connected device is often much larger, which requires the designer to carefully consider the security domain to ensure the product remains secure for its entire useful life.

The security domain is interesting for the designer because security is not something that can be bolted onto a system when it is nearly completed. Instead, security must be thought about as early as possible and built into the design. Plenty of tactical solutions for improving security exist, such as encrypting data and locking down a device's programming port. However, strategically, designers need to create layers of isolation within their applications. Then, each isolated layer in the design can be executed in its own security domain.

A designer may find it helpful to decompose their high-level application design into isolated regions that specifically call out the importance of the security context of where the various components are executing. Figure 2-13 shows an example of an updated high-level weather station design to demonstrate security isolation layers.

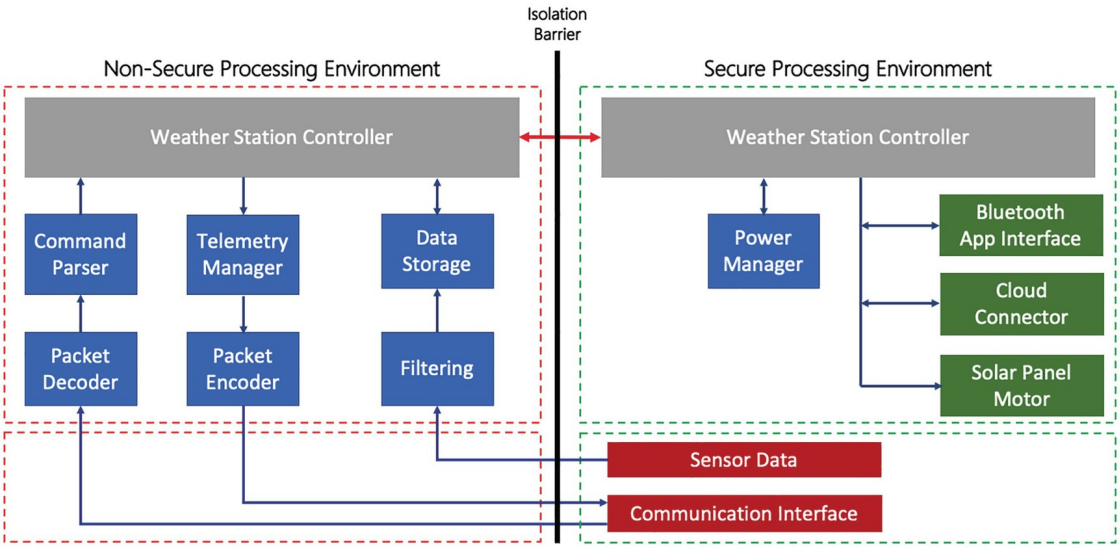

Figure 2-13 The top-level diagram is now broken up into execution domains in a high-level attempt to demonstrate isolation boundaries

It's interesting to note that Figure 2-13 doesn't specify which technology should be used for isolation at this point, only that there are secure processing environments (SPE) and nonsecure processing environments (NSPE). The solution could be to use a multicore processor with a dedicated security core or a single-core solution like Arm TrustZone. We will discuss designing secure applications in Chapter 3 and how designers develop their security requirements.

The Execution Domain

The third domain and the most interesting to most developers is the execution domain. Microcontroller applications have recently reached a point where the needs for performance and energy consumption cannot be achieved in single-core architecture. As a result, multicore microcontrollers are quickly becoming common in many applications. This drives the designer to carefully consider the design's execution domains for various components and features.

Multiple cores naturally force developers to decide which components and tasks run on which cores. For example, developers working with an ESP32, typically a dual-core Tensilica Xtensa LX6 32-bit processor, will often break their application up such that one core runs the Wi-Fi and Bluetooth stack. In contrast, the other core runs the real-time application. This ensures that the application is not choked by the Wi-Fi and Bluetooth stacks and can still achieve real-time requirements. An example domain partition between cores can be seen in Figure 2-14.

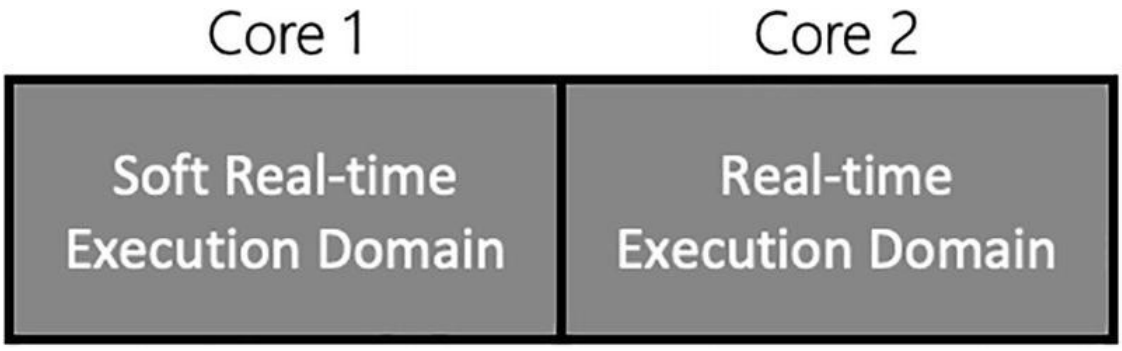

Figure 2-14 Multicore microcontrollers break an application into multiple execution domains that concurrently run on multiple cores. This is one example of execution domain decomposition

There are many potential use cases for multicore designers. We will ex-

plore the use cases and how to design applications that use multiple cores in Chapter 5.

Execution domain designs today go much further than simply deciding which core to run application components on. Cutting-edge systems and high-compute solutions such as machine learning may also contain several different execution domain types. For example, there may be soft and hard real-time microcontroller cores in addition to application processors, Tensor Processing Units (TPUs),[12] and Graphics Processing Units (GPUs).

Hybrid systems that combine processing elements are becoming much more common. For example, we are familiar with voice assistants such as Alexa, Siri HomePods, etc. These systems often employ a combination of low-power microcontroller cores for initial keyword spotting and then leverage an application processing core for relaying more complex processing to the cloud. These types of hybrid systems will become ubiquitous by the end of the 2020s.

The Cloud Domain

The final application domain to consider is the cloud domain. At first, this probably seems counterintuitive to an embedded designer. After all, doesn't the entire application live on the microcontroller? However, this may not necessarily be the case for a modern embedded system.

Today, designers can potentially offload computing from the embedded controller up to the cloud. For example, voice assistants don't have the processing power to classify all the word combinations that are possible. Instead, these systems are smart enough to recognize a single keyword like "Alexa" or "Hey Siri." Once the keyword is recognized, any remaining audio is recorded and sent to the cloud for processing. Once the cloud processes and classifies the recording, the result is sent back to the embedded system, which performs some action or response.

The cloud provides an application domain that embedded designers might usually overlook and should consider. However, the cloud isn't nec-

essarily a smoking gun to solve all computing problems because it can have a considerable impact in areas such as energy consumption, bandwidth, cost, and latency.

Final Thoughts

The software architecture is the blueprint, the road map that is used by developers to construct the designers' vision for the embedded software. Designers and developers must carefully decouple the hardware from the application's business logic. Embedded designers have multiple architectures that are available to them today that can help them avoid a giant ball of mud if they minimize coupling and maximize cohesion.

The software architecture in a modern system no longer has just a single domain to consider. Applications have become complex enough that multiple domains must be leveraged and traded against to successfully optimize a design. How the system is decomposed and which architecture will meet your needs depend on your system's end goals and requirements.

Action Items

To put this chapter's concepts into action, here are a few activities the reader can perform to start architecting their embedded software:

- Which approach to software architecture design do you use, top-down or bottom-up? What advantages and disadvantages do you see to the approach that you use? Is now the time to change your approach?
- What benefits can you gain by taking a top-down approach to architecture design? What difference would this make for you as an engineer and the business that employs you?
- Think through some of your recent projects. How would you rate the architecture in the following areas?
  - Coupling
  - Cohesion
  - Portability
  - Scalability
  - Flexibility

- Reusability

What changes do you need to make to your architecture to improve these characteristics?

- For a current or past project, what architecture did you use? What were the benefits and disadvantages of using that architecture? Are any of the architectures we discussed a better fit for your applications?
- Identify three changes you will make to your current architecture to better fit your team's needs.
- Have you considered the various application domains in your designs? What areas have you overlooked, and how could these have impacted the overall performance, security, and behavior of your applications? Take some time to break down a new application into its various application domains.

These are just a few ideas to go a little bit further. Carve out time in your schedule each week to apply these action items. Even minor adjustments over a year can result in dramatic changes!

Footnotes

1 https://en.wikipedia.org/wiki/IEEE_1471. IEEE 1471 has been superseded by IEEE 42010. (I just like the IEEE 1471 definition).

2 In a Q&A session, I was once accused of being a Rust advocate because I mentioned it! As of this writing, I still haven't written a single line of it, although it sounds intriguing.

3 https://encyclopedia2.thefreedictionary.com/abstraction+layer

4 A great article on change costs can be found at www.agilemodeling.com/essays/costOfChange.htm

5 We'd like to think that businesses don't fail, but 90% of businesses fail within their first ten years of operation!

6

https://bit.ly/33R6psu

7 https://en.wikipedia.org/wiki/Monolithic_application

8 CMSIS is Arm's Common Microcontroller Software Interface Standard.

9 The use of an RTOS with an OSAL will often require an OSAL extension to access unique RTOS features not supported in every RTOS.

10 www.guru99.com/microservices-tutorial.html

11 www.researchgate.net/figure/Armv7-M-operation-modes-and-privileged-levels_fig1_339784921

12 https://cloud.google.com/tpu/docs/tpus
