O'REILLY®

Second Edition

Making Embedded Systems

Design Patterns for Great Software

Early Release

RAW & UNEDITED

Elecia White

Making Embedded Systems

SECOND EDITION

Design Patterns for Great Software

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Elecia White

Making Embedded Systems

by Elecia White

Printed in the United States of America.

Published by O’Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

- Editors: Shira Evans and Brian Guerin
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- March 2024: Second Edition

Revision History for the Early Release

- 2023-06-23: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098151546 for release details.

978-1-098-15154-6

Chapter 1. Introduction

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at s*evans@oreilly.com*.

Embedded systems are different things to different people. To someone who has been working on servers, an application developed for a phone is an embedded system. To someone who has written code for tiny 8-bit microprocessors, anything with an operating system doesn't seem very embedded. I tend to tell nontechnical people that embedded systems are things like microwaves and automobiles that run software but aren't computers. (Most people recognize a computer as a general-purpose device.) Perhaps an easy way to define the term without haggling over technology is:

> *An embedded system is a computerized system that is purpose-built for its application.*

Because its mission is narrower than a general-purpose computer, an embedded system has less support for things that are unrelated to accomplishing the job at hand. The hardware often has constraints. For instance, consider a CPU that runs more slowly to save battery power, a system that uses less memory so it can be manufactured more cheaply, and processors that come only in certain speeds or support a subset of peripherals.

The hardware isn't the only part of the system with constraints. In some systems, the software must act deterministically (exactly the same each time) or in real time (always reacting to an event fast enough). Some systems require that the software be fault-tolerant with graceful degradation in the

face of errors. For example, consider a system in which servicing faulty software or broken hardware may be infeasible (such as a satellite or a tracking tag on a whale). Other systems require that the software cease operation at the first sign of trouble, often providing clear error messages (for example, a heart monitor should not fail quietly).

This short chapter goes over the high level view of embedded systems. Realistically, you could read the Wikipedia article but this is a way for us to get to know one another. Sadly, this chapter mostly talks about how difficult embedded systems are to develop. Between different compilers, debuggers, and resource constraints, the way we design and implement code is different from other varieties of software. Some might call the field a bit backward but that isn't true, we're focused on solving different problems, for the most part. And yet, there are some software engineering techniques that are useful but overlooked (but that's for the rest of the book).

One of the best things about embedded systems has been the maker movement. Everyone loves glowing lights so people get interested in making a career of the lower level software. If that is you, welcome. But I admit I'm expecting folks who have experience with hardware or software and need to know how to get the piece between them done well and efficiently.

At the end of every chapter, I have an interview question loosely related to the material. One of the leveling-up activities in my career was learning to interview other people for jobs on my team. Sorely disappointed that there isn't a resource on how to do that, I'm putting in my favorite interview questions and what I look for as the interviewer. They are a bit odd but I hope you enjoy them as much as I do.

Admittedly, I hope you enjoy all of embedded systems development as much as I do. There are challenges but that's the fun part.

Embedded Systems Development

Embedded systems are special, offering unique challenges to developers. Most embedded software engineers develop a toolkit for dealing with the constraints. Before we can start building yours, let's look at the difficulties associated with developing an embedded system. Once you become familiar with how your embedded system might be limited, we'll start on some principles to guide us to better solutions.

Compilers and Languages

Embedded systems use cross-compilers. Although a cross-compiler runs on your desktop or laptop computer, it creates code that does not. The cross-compiled image runs on your target embedded system. Because the code needs to run on your processor, the vendor for the target system usually sells a cross-compiler or provides a list of available cross-compilers to choose from. Many larger processors use the cross-compilers from the GNU family of tools.

Embedded software compilers often support only C, or C and C++. In addition, some embedded C++ compilers implement only a subset of the language (multiple inheritance, exceptions, and templates are commonly missing). There is a growing popularity for other languages but C and C++ remain the most prevalent.

Regardless of the language you need to use in your software, you can practice object-oriented design. The design principles of encapsulation, modularity, and data abstraction can be applied to any application in nearly any language. The goal is to make the design robust, maintainable, and flexible. We should use all the help we can get from the object-oriented camp.

Taken as a whole, an embedded system can be considered equivalent to an object, particularly one that works in a larger system (such as a remote control talking to a smart television, a distributed control system in a factory, an airbag deployment sensor in a car). In the higher level, everything is inherently object-oriented, and it is logical to extend this down into embedded software.

On the other hand, I don’t recommend a strict adherence to all object-oriented design principles. Embedded systems get pulled in too many directions to be able to lay down such a commandment. Once you recognize the trade-offs, you can balance the software design goals and the system design goals.

Most of the examples in this book are in C or C++. I expect that the language is less important than the concepts, so even if you aren’t familiar with the syntax, look at the code. This book won’t teach you any programming language (except for some assembly language), but good design principles transcend language.

Debugging

If you were to debug software running on a computer, you could compile and debug on that computer. The system would have enough resources to run the program and support debugging it at the same time. In fact, the hardware wouldn't know you were debugging an application, as it is all done in software.

Embedded systems aren't like that. In addition to a cross-compiler, you'll need a cross-debugger. The debugger sits on your computer and communicates with the target processor through the special processor interface (see Figure 1-1). The interface is dedicated to letting someone else eavesdrop on the processor as it works. This interface is often called JTAG (pronounced "jay-tag"), regardless of whether it actually implements that widespread standard.

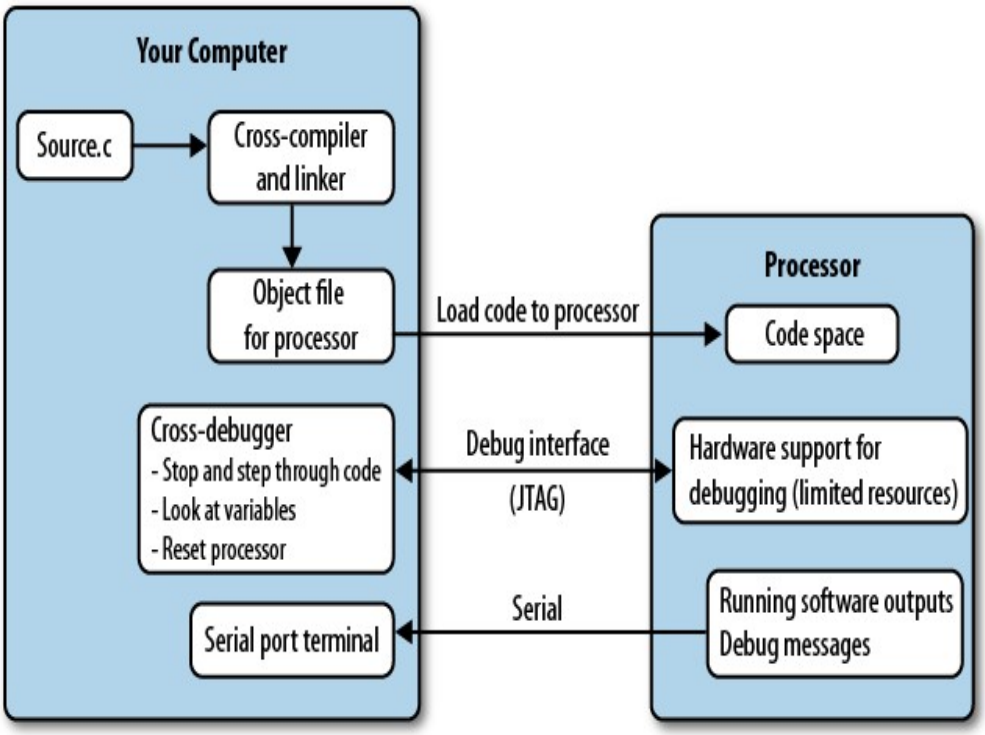


Figure 1-1. Computer and target processor

The processor must expend some of its resources to support the debug interface, allowing the debugger to halt it as it runs and providing the normal sorts of debug information. Supporting debugging operations adds cost to the processor. To keep costs down, some processors support a limited subset of features. For example, adding a breakpoint causes the processor to modify

the memory-loaded code to say “stop here.” However, if your code is executing out of flash (or any other sort of read-only memory), instead of modifying the code, the processor has to set an internal register (hardware breakpoint) and compare it at each execution cycle to the address being run, stopping when they match. This can change the timing of the code, leading to annoying bugs that occur only when you are (or maybe aren’t) debugging. Internal registers take up resources, too, so often there are only a limited number of hardware breakpoints available (frequently there are only two).

To sum up, processors support debugging, but not always as much debugging as you are accustomed to if you’re coming from the software world.

The device that communicates between your PC and the processor is generally called a hardware debugger, programmer, debug probe, in-circuit emulator (ICE), or JTAG adapter. These may refer (somewhat incorrectly) to the same thing, or they may be multiple devices. The debugger is specific to the processor (or processor family), so you can’t take the debugger you got for one project and assume it will work on another. The debugger costs add up, particularly if you collect enough of them or if you have a large team working on your system.

To avoid buying a debugger or dealing with the processor limitations, many embedded systems are designed to have their debugging done primarily via `printf` or some sort of lighter-weight logging to an otherwise unused communication port. Although incredibly useful, this can also change the timing of the system, possibly leaving some bugs to be revealed only after debugging output is turned off.

Writing software for an embedded system can be tricky, as you have to balance the needs of the system and the constraints of the hardware. Now you’ll need to add another item to your to-do list: making the software debuggable in a somewhat hostile environment, something we’ll talk more about in the next chapter.

Resource Constraints

An embedded system is designed to perform a specific task, cutting out the resources it doesn’t need to accomplish its mission. The resources under consideration include:

- Memory (RAM)
- Code space (ROM or flash)
- Processor cycles or speed
- Power consumption (which translates into battery life)
- Processor peripherals

To some extent, these are exchangeable. For example, you can trade code space for processor cycles, writing parts of your code to take up more space but run more quickly. Or you might reduce the processor speed in order to decrease power consumption. If you don't have a particular peripheral interface, you might be able to create it in software with I/O lines and processor cycles. However, even with trading off, you have only a limited supply of each resource. The challenge of resource constraints is one of the most pressing for embedded systems.

Another set of challenges comes from working with the hardware. The added burden of cross-debugging can be frustrating. During board bring-up, the uncertainty of whether a bug is in the hardware or software can make issues difficult to solve. Unlike your computer, the software you write may be able to do actual damage to the hardware. Most of all, you have to know about the hardware and what it is capable of. That knowledge might not be applicable to the next system you work on. You will need to learn quickly.

Once development and testing are finished, the system is manufactured, which is something most pure software engineers never need to consider. However, creating a system that can be manufactured for a reasonable cost is a goal that both embedded software engineers and hardware engineers have to keep in mind. Supporting manufacturing is one way you can make sure that the system that you created gets reproduced with high fidelity.

After manufacture, the units go into the field. With consumer products, that means they go into millions of homes where any bugs you created are enjoyed by many. With medical, aviation, or other critical products, your bugs may be catastrophic (which is why you get to do so much paperwork). With scientific or monitoring equipment, the field could be a place where the unit cannot ever be retrieved (or retrieved only at great risk and expense; consider the devices in volcano calderas), so it had better work. The life your

system is going to lead after it leaves you is something you must consider as you design the software.

After you've figured out all of these issues and determined how to deal with them for your system, there is still the largest challenge, one common to all branches of engineering: change. Not only do the product goals change, but the needs of the project also change through its lifespan. In the beginning, maybe you want to hack something together just to try it out. As you get more serious and better understand (and define) the goals of the product and the hardware you are using, you start to build more infrastructure to make the software debuggable, robust, and flexible. In the resource-constrained environment, you'll need to determine how much infrastructure you can afford in terms of development time, RAM, code space, and processor cycles. What you started building initially is not what you will end up with when development is complete. And development is rarely ever complete.

Creating a system that is purpose-built for an application has an unfortunate side effect: the system might not support change as the application morphs. Engineering embedded systems is not just about strict constraints and the eventual life of the system. The goal is figuring out which of those constraints will be a problem *later* in product development. You will need to predict the likely course of changes and try to design software flexible enough to accommodate whichever path the application takes. Get out your crystal ball.

Principles to Confront Those Challenges

Embedded systems can seem like a jigsaw puzzle, with pieces that interlock (and only go together one way). Sometimes you can force pieces together, but the resulting picture might not be what is on the box. However, we should jettison the idea of the final result as a single version of code shipped at the end of the project.

Instead, imagine the puzzle has a time dimension that varies over its whole life: conception, prototyping, board bring-up, debugging, testing, release, maintenance, and repeat. Flexibility is not just about what the code can do right now, but also about how the code can handle its lifespan. Our goal is to be flexible enough to meet the product goals while dealing with the resource constraints and other problems inherent to embedded systems.

There are some excellent principles we can take from software design to make the system more flexible. Using *modularity*, we separate the functionality into subsystems and hide the data each subsystem uses. With *encapsulation*, we create interfaces between the subsystems so they don't know much about each other. Once we have loosely coupled subsystems (or objects, if you prefer), we can change one area of software with confidence that it won't impact another area. This lets us take apart our system and put it back together a little differently when we need to.

Recognizing where to break up a system into parts takes practice. A good rule of thumb is to consider which parts can change independently. In embedded systems, this is helped by the presence of physical objects that you can consider. If a sensor X talks over a communication channel Y, those are separate things and good candidates for being separate subsystems (and code modules).

If we break things into objects, we can do some testing on them. I've had the good fortune of having excellent QA teams for some projects. In others, I've had no one standing between my code and the people who were going to use the system. I've found that bugs caught before software releases are like gifts. The earlier in the process errors are caught, the cheaper they are to fix and the better it is for everyone.

You don't have to wait for someone else to give you presents. Testing and quality go hand in hand. Writing test code for your system will make it better, provide some documentation for your code, and make other people think you write great software.

Documenting your code is another way to reduce bugs. It can be tough to know the level of detail when commenting your code.

```
i++; // increment the index
```

No, not like that. Lines like that rarely need comments at all. The goal is to write the comment for someone just like you, looking at the code a year from when you wrote it. By that time, future-you will probably be working on something different and have forgotten exactly what creative solution past-you came up with. Future-you probably doesn't even remember writing this code, so help yourself out with a bit of orientation (file and function headers). In general, though, assume the reader will have your brains and your general background, so document what the code does, not how it does it.

Finally, with resource-constrained systems, there is the temptation to optimize your code early and often. Fight the urge. Implement the features, make them work, test them out, and then make them smaller or faster as needed.

You have only a limited amount of time: focus on where you can get better results by looking for the bigger resource consumers *after* you have a working subsystem. It doesn't do you any good to optimize a function for speed if it runs rarely and is dwarfed by the time spent in another function that runs frequently. To be sure, dealing with the constraints of the system will require some optimization. Just make sure you understand where your resources are being used before you start tuning.

> *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."*
>
> Donald Knuth

Prototypes and Maker Boards

"But wait," you say, "I already have a working system built with an Arduino or Raspberry Pi Pico. I just need to figure out how to ship it."

I get it. The system does nearly everything you want it to do. The project seems almost done. Off-the-shelf development boards are amazing, especially the maker-friendly ones. They make prototyping easier. However, the prototype is not the product.

There are many tasks often forgotten at this stage. How will the firmware update? Does the system need to sleep to lower power consumption? Do we need a watchdog in case of catastrophic error? How much space and processing cycles do we need to save for future bug fixes and improvements? How do we manufacture many devices instead of one hand-built unit? How will we test for safety? Inexplicable user commands? Corner cases? Broken hardware?

Software aside, when the existing off-the-shelf boards don't meet your needs, custom boards are often necessary. The development boards may be too delicate, connected by fragile wires. They may be too expensive for your target market or take too much power. They may be the wrong size or shape.

They may not hold up under the intended environmental conditions (like temperature fluctuations in a car, getting wet, or going to space).

Whatever the reason for getting a custom board, it usually means removing the programming (and debugging) hardware that are part of processor development boards. At that point, you are looking at a system more resembling Figure 1-1.

Custom hardware will also push you out of some of the simplified development environments and software framework. Using the microprocessor vendor's hardware abstraction layers with a traditional compiler, you can get to smaller code sizes (often faster as well). The lack of anything between you and the processor allows you to create deterministic and real-time handling. You may also be able to use and configure an RTOS. You can more easily understand the licensing of the code you are using in the libraries.

Adding in an external programmer/debugger gives you debugging beyond printf, allowing you to see inside your code. This feels pretty magical after doing it the hard way for so long.

Still, there is a chasm between prototype and shipping device, between supporting one unit on your desk and a thousand or a million in the field. Don't be lulled into believing the project is complete because all of the features finally worked (one time, in perfect conditions).

Development boards and simplified development environments are great for prototypes and to help select a processor. But there will be a time when the device needs to be smaller, faster, or cheaper. At that point, the resource constraints kick in and you'll need a book like this one to help you.

Further Reading

There are many excellent references about design patterns. These two are my favorites:

- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.

There are many excellent references about design patterns, but this was the one that sparked the revolution. Due to its four collaborators, it is often known as the "Gang of Four" book (or a standard design pattern may be noted as a GoF pattern).

- Freeman, Eric T., Elisabeth Robson, Bert Bates, and Kathy Sierra. 2004. *Head First Design Patterns*. Cambridge, MA: O'Reilly Media.

For more information about getting from prototype to shipping units, I recommend Cohen, Alan. 2015. *Prototype to Product: A Practical Guide for Getting to Market*. Cambridge, MA: O'Reilly Media.

INTERVIEW QUESTION: HELLO WORLD

Here is a computer with a compiler and an editor. Please implement "hello world." Once you have the basic version working, add in the functionality to get a name from the command line. Finally, tell me what happens before your code executes—in other words, before the `main()` *function.* (Thanks to Phillip King for this question.)

In many embedded systems, you have to develop a system from scratch. With the first part of this task, I look for a candidate who can take a blank slate and fill in the basic functionality, even in an unfamiliar environment. I want him to have enough facility with programming that this question is straightforward.

This is a solid programming question, so you'd better know the languages on your resume. Any of them are fair game for this question. When I ask for a "hello world" implementation, I look for the specifics of a language (that means knowing which header file to include and using command arguments in C and C++). I want the interviewee to have the ability to find and fix syntax errors based on compiler errors (though I am unduly impressed when he can type the whole program without any mistakes, even typos).

NOTE

I am a decent typist on my own, but if someone watches over my shoulder, I end up with every other letter wrong. That's OK, lots of people are like that. So don't let it throw you off. Just focus on the keyboard and the code, not on your typing skills.

The second half of the question is where we start moving into embedded systems. A pure computer scientist tends to consider the computer as an imaginary ideal box for executing his beautiful algorithms. When asked what happens before the main function, he will tend to answer along the lines of "you know, the program runs," but with no understanding of what that implies.

However, if he mentions "start" or "cstart," he is well on his way in the interview. In general, I want him to know that the program requires initialization beyond what we see in the source, no matter what the platform is. I like to hear mention of setting the exception vectors to handle interrupts, initializing critical peripherals, initializing the stack, initializing variables, and if there are any C++ objects, calling creators for those. It is great if he can describe what happens implicitly (by the compiler) and what happens explicitly (in initialization code).

The best answers are step-by-step descriptions of everything that might happen, with an explanation of why they are important and how they happen in an embedded system. An experienced embedded engineer often starts at the vector table, with the reset vector, and moves from there to the power-on behavior of the system. This material is covered later in the book, so if these terms are new to you, don't worry.

An electrical engineer (EE) who asks this question gives bonus points when a candidate can, during further discussion of power-on behavior, explain why an embedded system can't be up and running 1 microsecond after the switch is flipped. The EE looks for an understanding of power sequencing, power ramp-up time, clock stabilization time, and processor reset/initialization delay.

Chapter 2. Creating a System Architecture

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at s*evans@oreilly.com*.

Even small embedded systems have so many details that it can be difficult to recognize where patterns can be applied. You'll need a good view of the whole system to understand which pieces have straightforward solutions and which have hidden dependencies. A good design is created by starting with an OK design and then improving on it, ideally before you start implementing it. And a system architecture diagram is a good way to view the system and start designing the software (or understanding the pile of code handed to you).

A product definition is seldom fixed at the start, so you may go round and round to hammer out ideas. Once the product functions have been sketched out on a whiteboard, you can start to think about the software architecture. The hardware folks are doing the same thing (hopefully in conjunction with you as the software designer, though their focus may be a little different). In short order, you'll have a software architecture and a draft schematic. Depending on your experience level, the first few projects you design will likely be based on other projects, so the hardware will be based on an existing platform with some changes.

In this chapter, we'll look at different ways of viewing the system in an effort to design better embedded software. In addition to better understanding the system, the goal of the diagrams is to identify ways to architect the software

to deal with changes: encapsulation, information hiding, having good interfaces between modules, and so on.

Embedded systems depend heavily on their hardware. Unstable hardware leaves the software looking buggy and unreliable. This is the place to start, making sure we can separate the changes in the hardware from bugs in the software.

It is possible (and usually preferable) to go the other way, looking at the system functions and determining what hardware is necessary to support them. However, I'm mostly going to focus on starting at the low-level hardware-interfacing software to reinforce the idea that your product depends on the hardware features being stable and accessible.

When you do a bottom-up design as described here, recognize that the hardware you are specifying is archetypal. Although you will eventually need to know the specifics of the hardware, initially accept that there will be some hardware that meets your requirements (i.e., some processor that does everything you need). Use that to work out the system, software, and hardware architectures before diving into details.

Getting Started

There are two ways to go about architecting systems. One is to start with a blank slate, with an idea or end goal but nothing in place. This is composition. Creating these system diagrams from scratch can be daunting. The blank page can seem vast.

The other way is to go from an existing product to the architecture, taking what someone else put together to understand the internals. Reverse engineering like this is decomposition, breaking things into smaller parts. When you join a team, sometimes you start with a close deadline, a bunch of code, and very little documentation with no time to write more.

Diagrams will help you understand the system. Your ability to see the overall structure will help you write your piece of the system in a way that keeps you on track to provide good-quality code and meet your deadline.

You may need to consider formulating two architectures: one local to the code you are working on and a more global one that describes the whole product so you can see how your piece fits in.

As your understanding deepens, your diagrams will get larger, particularly for a mature design. If the drawing is too complex, bundle up some boxes into one box and then expand on them in another drawing.

Some systems will make more sense with one drawing or another; it depends on how the original architects thought about what they were doing.

In the next few sections, I'm going to ask you to make some drawings of a system from the ground up, not looking at an existing system for simplicity's sake (all good systems will be more complicated than I can show here).

I'm showing them as sketches because that is what I want you to do. The different diagrams are ways of looking at the system and how it can be (should be? is?) built. We are not creating system documentation, instead these sketches are a way of thinking about the system from different perspectives.

Keep your sketches messy enough that if you have to do it all over again, it doesn't make you want to give up. Use a drawing tool if you'd like but pencil and paper is my personal favorite method. It lets me focus on the information, not the drawing method.

Creating System Diagrams

Just as hardware designers create schematics, you should make a series of diagrams that show the relationships between the various parts of the software. Such diagrams will give you a view of the whole system, help you identify dependencies, and provide insight into the design of new features.

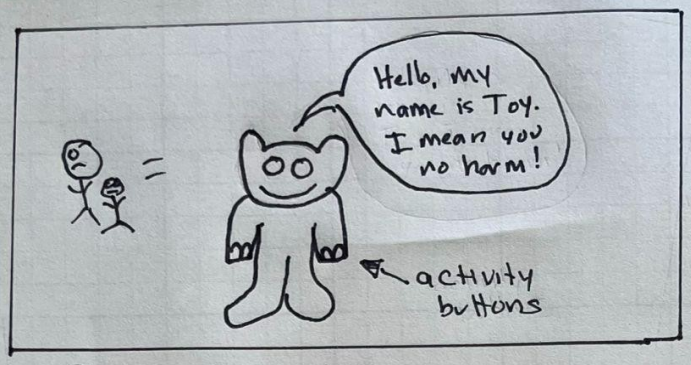
I recommend four types of diagrams:

- Context diagram
- Block diagram
- Organigram
- Layering diagram

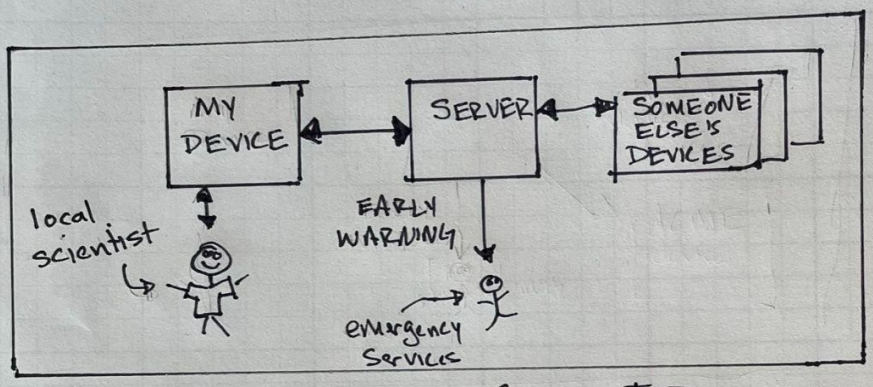
The Context Diagram

The first drawing is an overview of how your system fits into the world at large. If it is a children's toy, the diagram may be easy: a stick figure and

some buttons on the toy. If it is a device providing information to a network of seismic sensors with an additional output to local scientists, the system is more complicated, going beyond the piece you are working on. See Figure 2-1.



Simple Context Diagram



More Complex Context ~~Diagram~~ DIAGRAM

Description: Two part diagram in the sketch style of the next few Figures

Top Section: A large toy as creepy or uncanny as possible saying, "Hello, my name is Toy. I mean you no harm!" An arrow toward paw buttons saying "Activity buttons". A Daren and child fleeing (stick figures ok).

Bottom Section: A networked system with one device Marked "MY DEVICE" interfacing with a "local scientist" (in a white coat?). Another interface goes to a box marked "SERVER" which interfaces to several other devices (marked "SOMEONE ELSE'S DEVICES"). The Server also sends an "EARLY WARNING" to emergency services (identified by a stick fig in a fire fighter hat (or a firetruck or ~~a~~ only a fire fighter hat?)

Figure 2-1. Two context diagrams with different levels of complexity.

The goal here is a high-level context of how the system will be used by the customer. Don't worry about what is inside the boxes (including the one you are designing). The diagram should focus on the relationships between your device, users, servers, other devices, and other entities.

What problem is it solving? What are the inputs and outputs to your system? What are the inputs and outputs to the users? What are the inputs and outputs to the system-at-large? Focus on the use cases and world-facing interactions.

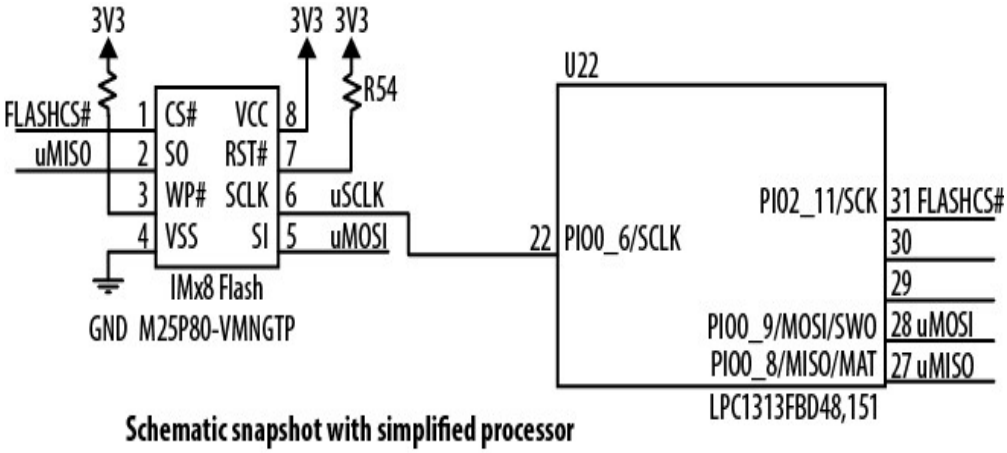
Ideally this type of diagram will help define the system requirements and envision likely changes. More realistically, it is a good way to remember the goals of the device as you get deeper into designing the software.

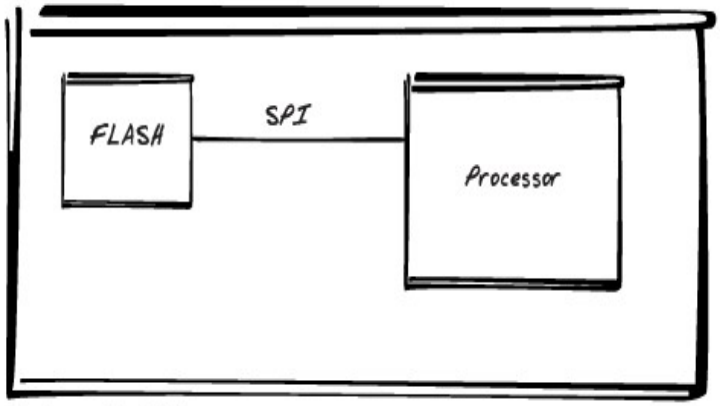
The Block Diagram

Design is straightforward at the start because you are considering the physical elements of the system, and you can think in an object-oriented fashion—no matter whether you are using an object-oriented language—to model your software around the physical elements. Each chip attached to the processor is an object. You can think of the wires that connect the chip to the processor (the communication methods) as another set of objects.

Start your design by drawing these as boxes where the processor is in the center, the communication objects are in the processor, and each external component is attached to a communication object.

For this example, I'm going to introduce a type of memory called flash memory. While the details aren't important here, it is a relatively inexpensive type of memory used in many devices. Many flash memory chips communicate over the SPI bus, a type of serial communication (discussed in more detail in Chapter 6). Most processors cannot execute code over SPI, so the flash is used for off-processor storage. Our schematic, shown at the top of Figure 2-2, shows that we have some flash memory attached to our processor via SPI. Don't be intimidated by the schematic! We'll cover them in more detail in the next chapter.





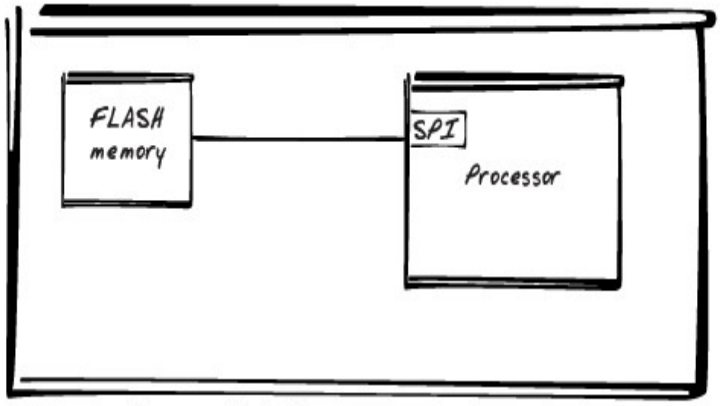


Figure 2-2. Comparison of schematic and initial software block diagram

Ideally, the hardware block diagram is provided by an electrical engineer along with the schematics to give you a simplified view of the hardware. If not, you may need to sketch your own on the way to a software block diagram.

In our software block diagram, we'll add the flash as a device (a box outside the processor) and a SPI box inside the processor to show that we'll need to write some SPI code. Our software diagram looks very similar to the hardware schematic at this stage, but as we identify additional software components, it will diverge.

The next step is to add a flash box inside the processor to indicate that we'll need to write some flash-specific code. It's valuable to separate the communication method from the external component; if there are multiple chips connected via the same method, they should all go to the same communications block in the chip. The diagram will at that point warn us to be extra careful about sharing that resource, and to consider the performance and resource contention issues that sharing brings.

Figure 2-2 shows a snippet of a schematic and the beginnings of a software block diagram. Note that the schematic is far more detailed. At this point in a design, we want to see the high-level items to determine what objects we'll need to create and how they all fit together. Keep the detailed pieces in mind (or on a different piece of paper), particularly where the details may have a system impact, but try to keep the details out of the diagram if possible. The goal is to break the system from a complete thing into bite-sized chunks you can summarize enough to fit in a box.

The next step is to add some higher-level functionality. What is each external chip used for? This is simple when each has only one function. For example, if our flash is used to store bitmaps to put on a screen, we can put a box on the architecture to represent the display assets. This doesn't have an off-chip element, so its box goes in the processor. We'll also need boxes for a screen and its communication method, and another box to convey flash-based display assets to the screen. It is better to have too many boxes at this stage than too few. We can combine them later.

Add any other software structures you can think of: databases, buffering systems, command handlers, algorithms, state machines, etc. You might not know exactly what you'll need for those (we'll talk about some of them in more detail later in the book), but try to represent everything from the hardware to the product function in the diagram. See Figure 2-3.

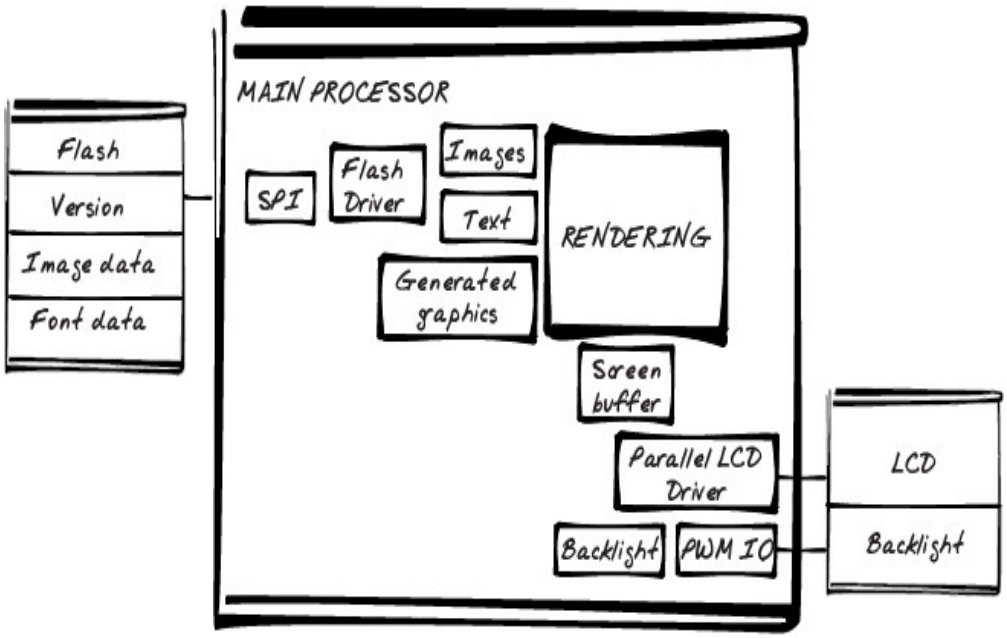


Figure 2-3. More detailed software block diagram

After you have sketched out this diagram on a piece of paper or a whiteboard (probably multiple times because the boxes never end up being big enough or in the right place), you may think you've done enough. However, other types of drawings often will give additional insight.

NOTE

If you are trying to understand an existing code base, get a list of the code files. If there are too many to count, use the directory names and libraries. Someone decided that these are modules, and maybe they even tried to consider them as objects, so they go in our diagram as boxes.

Where to put the boxes and how to order them is a puzzle, and one that may take a while to figure out.

Looking at other types of drawing may show you some hidden, ugly spots with critical bottlenecks, poorly understood requirements, or an innate failure to implement the product's features on the intended platform. Often these deficiencies can be seen from only one view or are represented by the boxes that change most between the different diagrams. By looking at them from the right perspective, ideally you will identify the tricky modules and also see a path to a good solution.

Organigram

The next type of software architecture diagram looks like an organizational chart (organigram). In fact, I want you to think about it as an org chart. Like a manager, the upper level components tell the lower ones what to do. Communication is not (should not be!) one direction. The lower level pieces will act on the orders to the best of their ability, provide requested information, and notify its boss when errors arise. Think of the system as a hierarchy and this diagram shows the control and dependencies.

Figure 2-4 shows discrete components and which ones call others. The whole system is a hierarchy, with `main` at the highest level. If you have the device's algorithm planned out and know how the algorithm is going to use each piece, you can fill in the next level with algorithm-related objects. If you don't think they are going to change, you can start with the product-related features and then drop down to the pieces we do know, putting the most complex on top. Next, fill in the lower-level objects that are used by a higher-level object. For instance, our SPI object is used by our flash object, which is used by our display assets object, and so on. You may need to add some pieces here, ones you hadn't thought of before. You should determine whether those pieces need to go on the block diagram, too (probably).

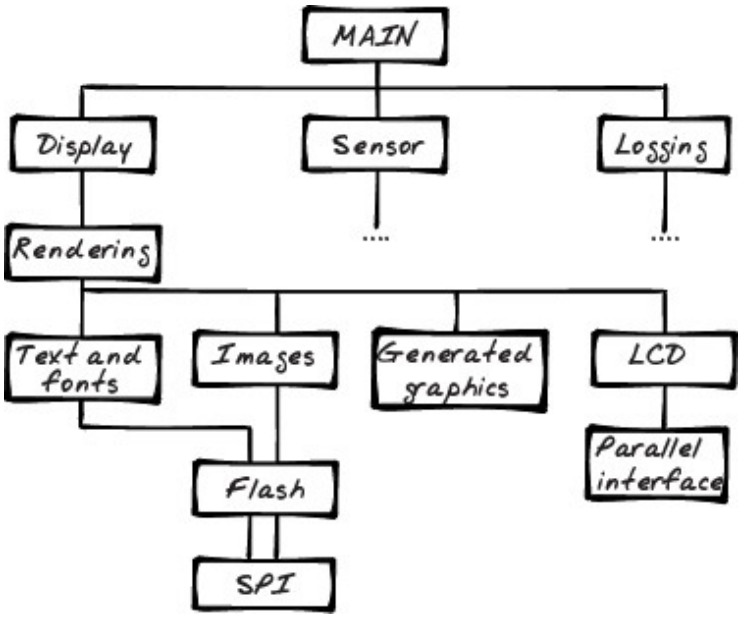


Figure 2-4. Organizational diagram of software architecture

However, as much as we'd like to devote a component to one function (e.g., the display assets in the flash memory), the limitations of the system (cost, speed, etc.) don't always support that. Often you end up cramming multiple

not particularly compatible functions into one component or communication pathway.

In the diagram, you can see where the text and images share the flash driver and its child SPI driver. This sharing is often necessary, but it is a red flag in the design because you'll need special care to avoid contention around the resource and make sure it is available when needed. Luckily, this figure shows that the rendering code controls both and will be able to ensure that only one of the resources—text or images—is needed at a time, so a conflict between them is unlikely.

Let's say that your team has determined that the system we've been designing needs each unit to have a serial number. It is to be programmed in manufacturing and communicated upon request. We could add another memory chip as a component, but that would increase cost, board complexity, and the software complexity. The flash we already have is large enough to fit the serial number. This way, only software complexity has to increase. (Sigh.)

In Figure 2-5, we print the system serial number through the flash that previously was devoted to the display assets. If the logging subsystem needs to get the serial number asynchronously from the display assets (say I have two threads or my display assets are used in an interrupt), the software will have to avoid collisions and any resulting corruption.

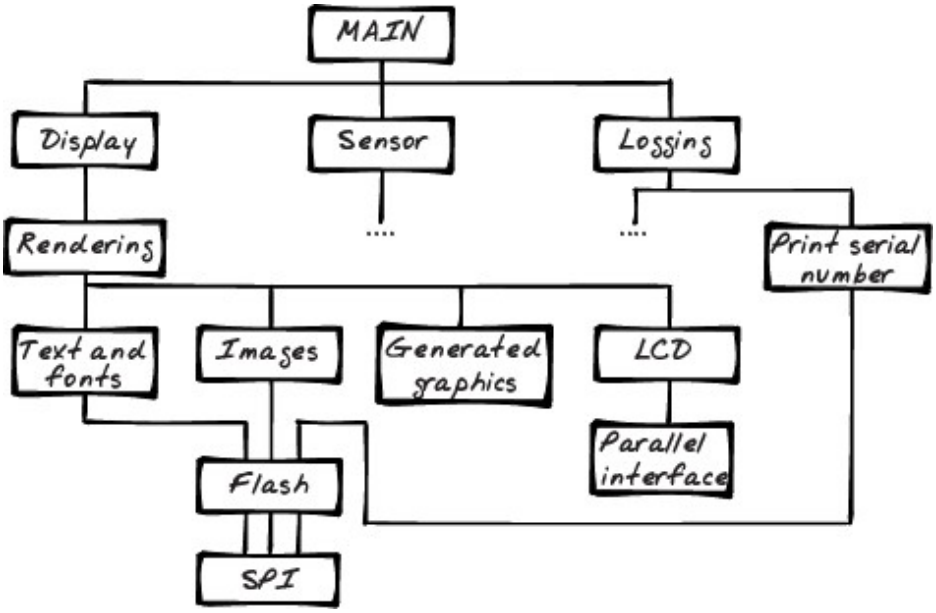


Figure 2-5. Organizational diagram with a shared resource

Each time something like this is added, some little piece where you are using A and B and have to consider a potential interaction with C, the system becomes a little less robust. This added awareness is very hard to document, and shared resources cause pains in the design, implementation, and maintenance phases of the project. The example here is pretty easily fixed with a flag, but all shared resources should make you think about the consequences.

Returning to the org chart metaphor, people managed by two entities will have conflicting instructions and priorities. While best to avoid the situation, sometimes you need to manage yourself a bit to handle the complexities of the job.

NOTE

If you are trying to understand an existing code base, the organigram can be constructed by walking through the code in a debugger. You start in main() and step into interesting functions, trying not to get too lost in the details.

It might take a few repetitions, but the goal is to get a feel for where the flow of code is going and how it gets there.

Layering Diagram

The last architecture drawing looks for layers and represents objects by their estimated size, as in Figure 2-6. This is another diagram to start with paper and pencil. Start at the bottom of the page and draw boxes for the things that go off the processor (such as our communication boxes). If you anticipate that one is going to be more complicated to implement, make it a little larger. If you aren't sure, make them all the same size.

Next, add to the diagram the items that use the lowest layer. If there are multiple users of a lower-level object, they should all touch the lower-level object (this might mean making the lower-level component bigger). Also, each object that uses something below it should touch all of the things it uses, if possible.

That was a little sneaky. I said to make the object size depend on its complexity. Then I said that if an object has multiple users, make it bigger. As described in the previous section, shared resources increase complexity.

So when you have a resource that is shared by many things, its box gets bigger so it can touch all of the upper modules. This reflects an increase in complexity, even if the module seems straightforward. It isn't only bottom layers that have this problem. In the diagram, I initially had the rendering box much smaller because moving data from the flash to the LCD is easy. However, once the rendering box had to control all the bits and pieces below it, it became larger. And sure enough, on the project that I took this snippet from, ultimately rendering became a relatively large module and then two modules.

Eventually, the layering diagram shows you where the layers in your code are, letting you bundle groups of resources together if they are always used together. For example, the LCD and parallel I/O boxes touch only each other. If this is the final diagram, maybe those could be combined to make one module. The same goes for the backlight and PWM output.

Also look at the horizontal groupings. Fonts and images share their higher-level and lower-level connections. It's possible they should be merged into one module because they seem to have the same inputs and outputs. The goal of this diagram is to look for such points and think about the implications of combining the boxes in various ways. You might end up with a simpler design.

Finally, if you have a group of several modules that try to touch the same lower-level item, you might want to take some time to break apart that resource. Would it be useful to have a flash driver that dealt only with the serial number? Maybe one that read the serial number on initialization and then never reread it, so that the display subsystem could keep control of the flash? Understand the complexity of your design and your options for designing the system to modify that complexity. A good design can save time and money in implementation and maintenance.

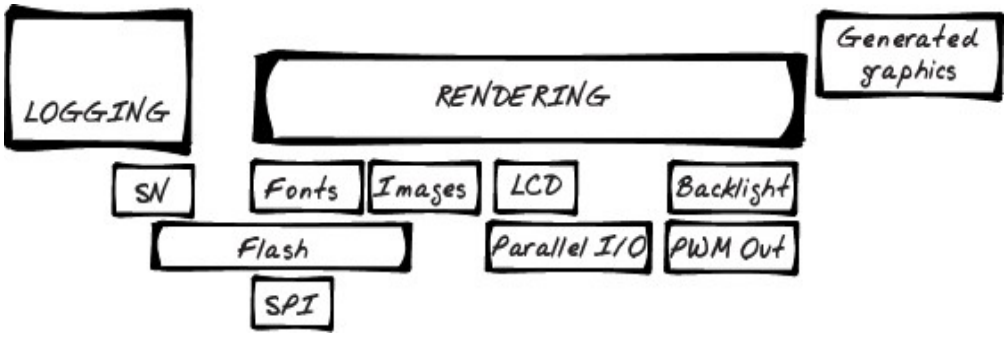


Figure 2-6. Software architecture layering diagram

Design for Change

So now, sitting with the different architecture drawings, where do you go next? Maybe you've realized there are a few pieces of code you didn't think about initially. And maybe you have progressed a bit at figuring out how the modules interact. Before we consider those interactions (interfaces), there is one thing that is really worth spending some time on: *what is going to change?* At this stage everything is experimental, so it is a good bet that any piece of the system puzzle is going to change. However, the goal is to harden your initial architecture (and code) against the possible changes to the system features or actual hardware.

Given your product requirements, you may have confidence that some features of your system won't change. Our example, whatever it does, needs a display, and the best way to send bitmaps to it seems like flash. Many flash chips are SPI, so that seems like a good bet, too. However, exactly which flash chip is used will likely change. The LCD, the image, or font data also may change. Even the way you store the image or font data might change. The boxes in the diagram should represent the Platonic ideals of each thing instead of a specific implementation.

Encapsulate Modules

The diagramming process leads to interfaces that don't depend specifically on the content or behavior of the modules that they connect (this is encapsulation!). We use the different architecture drawings to figure out the best places for those interfaces. Each box will probably have its own interface. Maybe those can be mashed together into one object. But is there a good reason to do it now instead of later?

Sometimes, yes. If you can reduce some complexity of your diagrams without giving up too much flexibility in the future, it may be worth collapsing some dependency trees. Here are some things to look for:

- In the organigram, look for objects that are used by only one other object. Are these both fixed? If they are unlikely to change or are likely to change together as you evolve the

system, consider combining them. If they change independently, it is not a good area to consider for encapsulation.

- In the layering diagram, look for collections of objects that are always used together. Can these be grouped together in a higher-level interface to manage the objects? You'd be creating a hardware abstraction layer.
- Which modules have lots of interdependencies? Can they be broken apart and simplified? Or can the dependencies be grouped together?
- Can the interfaces between vertical neighbors be described in a few sentences? That makes it a good place for encapsulation to help you create an interface (for others to reuse your code or just for ease of testing).

In the example system, the LCD is attached to the parallel interface. In every diagram, it is simple, with no additional dependencies and no other subsystem requiring access to the parallel interface. You don't need to expose the parallel interface; you can encapsulate (hide) it within the LCD module.

Conversely, imagine the system without the rendering module to encapsulate the display subsystem. The layering diagram might show this best (Figure 2-5). The fonts, images, LCD, and backlight would all try to touch each other, making a mess of the diagram.

Each box you're left with is likely to become a module (or object), look at the drawing. How can you make it simpler? Encapsulation in the software makes for a cleaner architecture drawing, and the converse is true as well.

Delegation of Tasks

The diagrams also help you divide up and apportion work to minions. Which parts of the system can be broken off into separate, describable pieces that someone else can implement?

Too often we want our minions to do the boring part of the job while we get to do all of the fun parts. ("Here, minion, write my test code, do my documentation, fold my laundry.") Not only does this drive away the good

minions, it tends to decrease the quality of your product. Instead, think about which whole box (or whole subtree) you can give to someone else. As you try to describe the interdependencies to your imaginary minion, you may find that they become worse than you've represented in the diagrams. Or you may find that a simple flag (such as a semaphore) to describe who currently owns the resource may be enough to get started.

NOTE

What if you have no chance of getting minions? It is still important to go through this thought process. You want to reduce interdependencies where possible, which may cause you to redesign your system. And where you can't reduce the interdependencies, you can at least be wary of them when coding.

Looking for things that can be split off and accomplished by another person will help you identify sections of code that can have a simple interface between them. Also, when marketing asks how this project could be made to go faster, you'll have an answer ready for them. However, there is one more thing our imaginary minion provides: assume he or she is slightly deficient and you need to protect yourself and your code from the faulty minion's bad code.

What sort of defensive structures can you picture erecting between modules? Imagine the data being passed between modules. What is the minimum amount of data that can move between boxes (or groups of boxes)? Does adding a box to your group mean that significantly less data is passed? How can the data be stored in a way that will keep it safe and usable by all those who need it?

The minimization of complexity between boxes (or at least between groups of boxes) will make the project go more smoothly. The more that your minions are boxed into their own sets of code, with well-understood interfaces, the more everyone will be able to test and develop their own code.

Driver Interface: Open, Close, Read, Write, IOCTL

The previous section used a top-down view of module interfaces to train you to consider encapsulation and think about where you can get help from

another person. Going from the bottom up works as well. The bottom here consists of the modules that talk to the hardware (the drivers).

NOTE

Top-down design is when you think about what you want and then dig into what you need to accomplish your goals.

Bottom-up design is when you consider what you have and build what you can out of that.

Usually I end up using a combination of the two: yo-yo design.

Many drivers in embedded systems are based on an API used to call devices in Unix systems. Why? Because the model works well in many situations and saves you from reinventing the wheel every time you need access to the hardware. The interface to Unix drivers is straightforward:

`open`

: Opens the driver for use. Similar to (and sometimes replaced by) `init`.

`close`

: Cleans up the driver, often so another subsystem can call `open`.

`read`

: Reads data from the device.

`write`

: Sends data to the device.

`ioctl` *(Pronounced "eye-octal")*

: Stands for input/output (I/O) control and handles the features not covered by the other parts of the interface. It is somewhat discouraged by kernel programmers due to its lack of structure, but it is still very popular.

In Unix, a driver is part of the kernel. Each of these functions takes an argument with a file descriptor that represents the driver in question (such as `/dev/tty01` for the first terminal on the system). That gets pretty cumbersome for an embedded system without an operating system. The idea

is to model your driver upon Unix drivers. A sample functionality for an embedded system device might look like any of these:[1]

- `spi.open()`
- `spi_open()`
- `SpiOpen(WITH_LOCK)`
- `spi.ioctl_changeFrequency(THIRTY_MHz)`
- `SpiIoctl(kChangeFrequency, THIRTY_MHz)`

This interface straightens out the kinks that can happen at the driver level, making them less specific to the application and creating reusable code. Further, when others come up to your code and the driver looks like it has these functions, they will know what to expect.

NOTE

The driver model in Unix sometimes includes two newer functions. The first, `select` (or `poll`), waits for the device to change state. That used to be done by getting empty reads or polling `ioctl` messages, but now it has its own function. The other one is `mmap`, which controls the memory map the driver shares with the code that calls it.

If your round peg can't fit into this POSIX-compliant square hole, don't force it. But if it looks like it might, starting with this standard interface can make your design just a little better and easier to maintain.

Adapter Pattern

One traditional software design pattern is called *adapter* (or sometimes *wrapper*). It converts the interface of an object into one that is easier for a client (a higher-level module). Often, adapters are written over software APIs to hide ugly interfaces or libraries that change.

Many hardware interfaces are like ungainly software interfaces. That makes each driver an adapter, as shown in Figure 2-7. If you create a common interface to your driver (even if it isn't `open`, `close`, `read`, `write`, `ioctl`), the hardware interface can change without your upper-level software changing.

Ideally, you can switch platforms altogether and need only to rework the underpinnings.

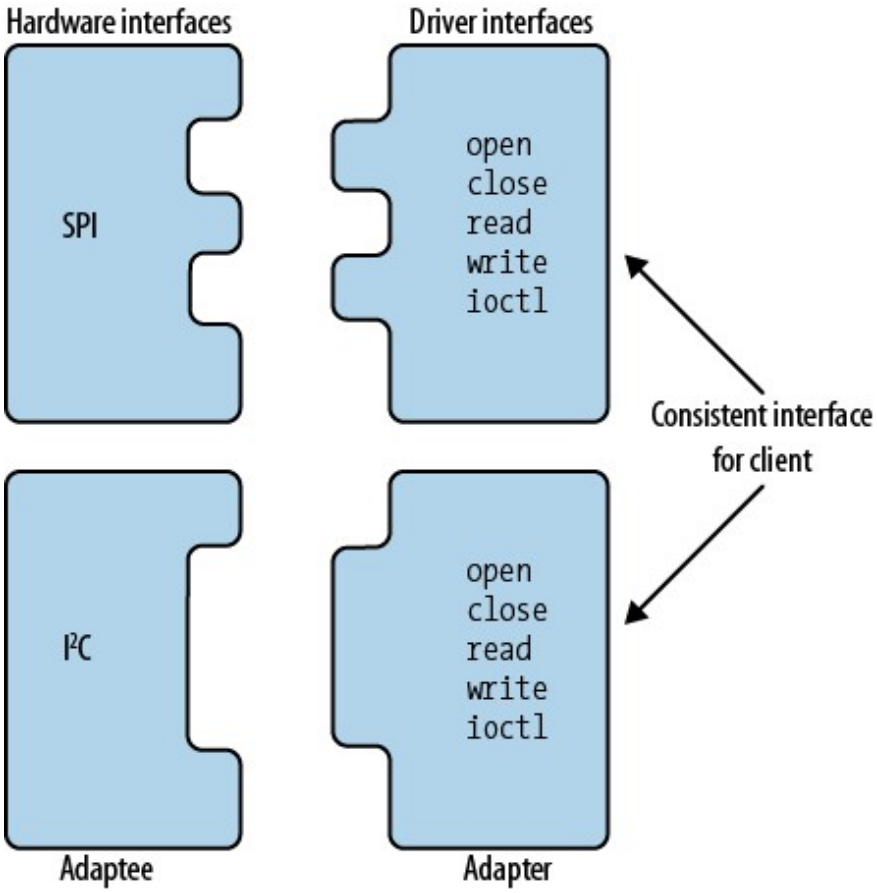


Figure 2-7. Drivers implement the adapter pattern

Note that drivers are stackable, as shown in Figure 2-7. In our example we have a display that uses flash memory, which in turn uses SPI communication. When you call `open` for the display, it will call its subsystems initialization code, which will call `open` for the flash, which will call `open` for the SPI driver. That's three levels of adapters, all in the cause of portability and maintainability.

The layers and the adapters add complexity to the implementation. They may also require more memory or add delays to your code. That is a tradeoff. You can have simpler architectures with all of the good maintainability, testing, portability, and so on for a small price. Unless your system is very resource constrained, this is usually a good trade.

If the interface to each level is consistent, the higher-level code is pretty impervious to change. For example, if our SPI flash changes to an I2C

EEPROM (a different communication bus and a different type of memory), the display driver might not need to change, or it might only need to replace flash functions with EEPROM ones.

In Figure 2-8, I've added a function called `test` to the interface of each of the modules. In Chapter 3, I'll discuss some strategies for choosing automated tests to make your code more robust. For now, they are just placeholders.

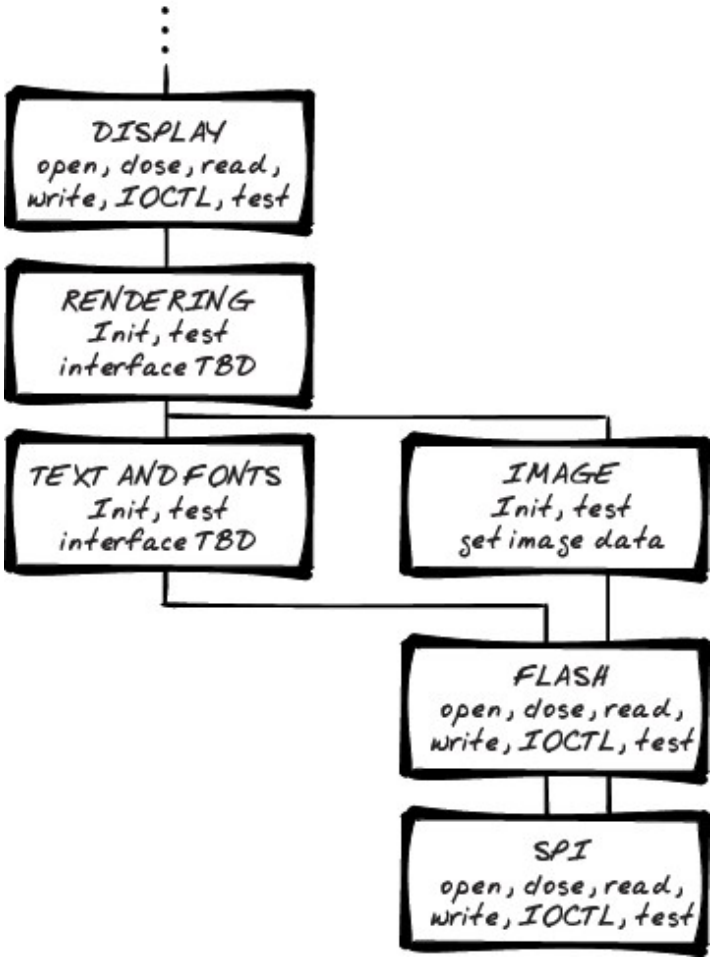


Figure 2-8. Interfaces for display subsystem and below

Creating Interfaces

Moving away from drivers, your definition of a module interface depends on the specifics of the system. It is pretty safe to say that most modules will also need an initialization function (though drivers often use `open` for this). Initialization may happen as objects are instantiated during startup, or it may be a function called as the system initializes. To keep modules encapsulated (and more easily reused), high-level functions should be responsible for

initializing the modules they depend upon. A good `init` function should be able to be called multiple times if it is used by different subsystems. A very good `init` function can reset the subsystem (or hardware resource) to a known good state in case of partial system failure.

Now that you don’t have a completely blank slate anymore, it may be easier to fill in the interface of each of your boxes. Consider how to retain encapsulation for your module, your hypothetical minion’s contribution, and the driver model, then start working out the responsibilities of each box in your architecture diagrams.

NOTE

Having created different views of the architecture, you probably won’t want to maintain each one. As you fill in the interface, you may want to focus on whichever one is most useful to you (or clearest to your boss).

Example: A Logging Interface

The goal of the logging module is to implement a robust and reusable logging system. In this section we’ll start off by defining the requirements of the interface, and then explore some options for the interface (and the local memory). It doesn’t matter what your communication method is. By coding to the interface in the face of limitations, you leave yourself open to reusing your code in another system.

NOTE

Logging debug output can slow down a processor significantly. If your code behavior changes when you turn logging on and off, consider how the timing of various subsystems works together.

The implementation is dependent on your system. Sometimes the best you can do is toggle an I/O line attached to an LED and send your messages via Morse code (I kid you not). However, most of the time you get to write text debug messages to some interface. Making a system debuggable is part of making it maintainable. Even if your code is perfect, the person who comes

after you might not be so lucky when they add a newly required feature. A logging subsystem will not only help you during development, it is an invaluable tool during maintenance.

A good logging interface hides the details of how the logging is actually accomplished, letting you hide changes (and complexity). Your logging needs may change with the development cycle. For example, in the initial phase, your development kit may come with an extra serial port and your logging information can go to a computer. In a later phase, the serial port might not be available, so you might have to pare your output back to an LED or two.

I have occasionally wished for a complete mind meld when a system is acting oddly. Sadly, there are never enough resources available to output everything you might want. Further, your logging methodology may change as your product develops. This is an area where you should encapsulate what changes by hiding its functions called from the main interface. The small overhead you add will allow greater flexibility. If you can code to the interface, changing the underlying pathway won't matter.

Commonly, you want to output a lot of information over a relatively small pipe. As your system gets bigger, the pipe looks smaller. Your pipe may be a simple serial port connecting via RS-232 to your computer, a pathway available only with special hardware. It may be a special debug packet that happens over your network. The data may be stored in an external RAM source and be readable only when you stop the processor and read via JTAG. The log may be available only when you run on the development kit, and not on the custom hardware. So the first big requirement for the module is this: the logging interface should be able to handle different underlying implementations.

Second, as you work on one area of the system at a time, you may not need (or want) messages from other areas. So the requirement is that logging methods should be subsystem-specific. Of course, you do need to know about catastrophic issues with other subsystems.

The third requirement is a priority level that will let you debug the minutiae of the subsystem you are working on without losing critical information from other parts of the code.

From requirements to an interface

Defining the main interface requirements of a module is often enough of a definition, particularly during design. However, those are a lot of words for something that can be summed up in one line of code:

```
void Log(enum eLogSubSystem sys, enum eLogLevel level, char *msg);
```

This prototype isn't fixed in stone and may change as the interface develops, but it provides a useful shorthand to other developers.

The log levels might include: none, information only, debugging, warning, error, and critical. The subsystems will depend on your system but might include: communications, display, system, sensor, updating firmware, etc.

Note that the log message takes a string, unlike `printf` or `iostream`, which take variable arguments. You can always use a library to build the message if you have that functionality. However, the `printf` and `iostream` family of functions are some of the first things cut from a system that needs more code space and memory. If that is the case, you'll probably end up implementing what you need most on your own, so this interface should have the bare minimum of what you'll need. In addition to printing strings, usually you'll need to be able to print out at least one number at a time:

```
void LogWithNum(enum eLogSubSystem sys, enum eLogLevel level, char *msg, int number);
```

Using the subsystem identifiers and the priority levels allows you to change the debug options remotely (if the system allows that sort of thing). When debugging something, you might start with all subsystems set to a verbose priority level (e.g., debug), and once a subsystem is cleared, raise its priority level (e.g., error). This way you get the messages you want when you want them. So we'll need an interface to allow this flexibility on a subsystem and priority level:

```
void LogSetOutputLevel(enum eLogSubSystem sys, enum eLogLevel level)
```

Because the calling code doesn't care about the underlying implementation, it shouldn't have direct access to it. All logging functions should go through this log interface. Ideally, the underlying interface isn't shared with any other modules, but your architecture diagram will tell you if that isn't true. The log initialization function should call whatever it depends on, whether it's to initialize a serial port driver or set up I/O lines.

Because logging can change the system timing, sometimes the logging system needs to be turned off in a global way. This allows you to say with some certainty that the debugging subsystem is not interfering in any way

with any other part of the code. Although these might not be used often, an on/off switch is an excellent addition to the interface:

```
void LogGlobalOn();
void LogGlobalOff();
```

VERSION YOUR CODE

At some point, someone will need to know exactly what revision of code is running. In the application world, putting a version string in the help/about box is straightforward. In the embedded world, the version should be available via the primary communication path (USB, WiFi, UART or other bus). If possible, this should print out automatically on boot. If that is not possible, try to make it available through a query. If that is not possible, it should be compiled into its own object file and located at a specific address so that it is available for inspection.

The ideal version is of the form A.B.C, where:

- A is the major version (1 byte)
- B is the minor version (1 byte)
- C is a build indicator (2 bytes)

If the build indicator does not increment automatically, you should increment it often (numbers are cheap). Depending on your output method and the aesthetics of your system, you may want to add an interface to your logging code for displaying the version properly:

void LogVersion(struct sFirmwareVersion *v)

The runtime code is not the only piece of the system that should have a version. Each piece that gets built or updated separately should have a version that is part of the protocol. For example, if an EEPROM is programmed in manufacturing, it should have a version that is checked by the code before the EEPROM is used. Sometimes you don’t have the space or processing power to make your system backward compatible, but it is critical to make sure all of the moving pieces are currently compatible.

Designs of the other subsystems do not (and should not) depend on the way that logging is carried out. If you can say that about the interface to your modules ("The other subsystems do not depend on the way XYZ is carried out; they just call the given interface"), you've successfully designed the interfaces of the system.

State of logging

As you work on the architecture, some areas will be easier to define than others, especially if you've done something similar before. And as you define interfaces, you may find that ideas come to you and implementation will be easy (even fun), but only if you start *right now*.

Hold back a bit when you get this urge to jump into implementation; try to keep everything to the same level. If you finish gold-plating one module before defining the interface to another, you may find that they don't fit together very well.

Once you've gotten a little further with all the modules in the system, it may be time to consider the state associated with each module. In general, the less state held in the module, the better (so that your functions do the same things every time you call them). However, eliminating all state is generally unavoidable (or at least extremely cumbersome).

Back to our logging module: can you see the internal state needed? The `LogGlobalOn` and `LogGlobalOff` functions set (and clear) a single variable. `LogSetOutputLevel` needs to have a level for each subsystem.

You have some options for how to implement these variables. If you want to eliminate local state, you could put them in a structure (or object) that every function calling into the logging module would need to have. However, that means passing the logging object to every function that might possibly need logging and passing it to every function that has a function that needs to log something.

NOTE

You may think passing around the state like that is convoluted. And for logging, I'd agree. However, have you ever wondered what is in the file handler you get back when your code opens a file? Open files embody lots of state information including the current read and write positions, a buffer, and a handle to where the file is on the drive.

Maybe passing all these parameters around isn't a good idea. How can every user of the logging subsystem get access to it? As noted in the sidebar "Object-Oriented Programming in C", with a little extra work, even C can create objects that make this parameter-passing problem simpler. Even in a more object-oriented language, you could have a module where the functions are globally available and the state is kept in a local object. However, there is another way to give access to the logging object without opening up the module completely.

OBJECT-ORIENTED PROGRAMMING IN C

Why not use C++? Most systems have pretty good embedded C++ compilers. However, there is a lot of code already written in C, and sometimes you need to match what is already there. Or maybe you just like C for its speed. That doesn't mean you should leave your object-oriented principles behind.

One of the most critical ideas to retain is data hiding. In an object-oriented language, your object (class) can contain private variables. This is a polite way of saying they have internal state that no one else can see. C has different kinds of global variables. By scoping a variable appropriately, you can mimic the idea of private variables (and even friend objects). First, let's start with the C equivalent of a public variable, declared outside a function, usually at the top of your C file:

```
// everyone can see this global with an "extern tBoolean_t gLogOnPublic;" in the
// file or in the header
tBoolean gLogOnPublic;
```

These are the global variables upon which spaghetti code is built. Try to avoid them. A private variable is declared with the `static` keyword and is declared outside a function, usually at the top of your C file.

```
// file variables are globals with some encapsulation
             static tBoolean gLogOnPrivate;
```

WARNING

The `static` keyword means different things in different places; it's actually kind of annoying that way. For functions and variables outside functions, the keyword means "hide me so no one other files can see" and limits scope. For variables within a function, the `static` keyword maintains the value between calls, acting as a global variable whose scope is only the function it is declared in.

A set of loose variables is a little difficult to track, so consider a structure filled with the variables private to a module:

```
// contain all the global variables into a structure:
struct {
  tBoolean logOn;
  enum eLogLevel outputLevel[NUM_LOG_SUBSYSTEMS];
} sLogStruct;
static struct sLogStruct gLogData;
```

If you want your C code to be more like an object, this structure would not be part of the module, but would be created (`malloc`'d) during initialization (`LogInit`, for the logging system discussed in this chapter) and passed back to the caller like this:

```
struct sLogStruct* LogInit(){
  int i;
  struct sLogStruct *logData = malloc(sizeof(*logData));
  logData->logOn = FALSE;
  for (i=0; i < NUM_LOG_SUBSYSTEMS; i++) {
    logData-> outputLevel[i] = eNoLogging;
  }
  return logData;
}
```

The structure can be passed around as an object. Of course, you'd need to add a way to free the object; that just requires adding another function to the interface.

Pattern: Singleton

Another way to make sure every part of the system has access to the same log object is to use another design pattern, this one called *singleton*.

When it is important that a class has exactly one instance, the singleton pattern is commonly seen. In an object-oriented language, the singleton is responsible for intercepting requests to create new objects in order to preserve its solitary state. The access to the resource is global (anyone can use it), but all accesses go through the single instance. There is no public constructor. In C++ this would look like:

```
class Singleton {
public:
  static Singleton* GetInstance() {
    if (instance_ == NULL) {
      instance_ = new Singleton;
    }
    return instance_;
  }
```

```
protected:
  Singleton(); // no one can create this except itself

private:
  static Singleton* instance_; // the one single instance
};

// Define the one and only Singleton pointer
Singleton* Singleton::instance_ = NULL;
```

For logging, the singleton lets the whole system have access to the system with only one instantiation. Often when you have a single resource (such as a serial port) that different parts of the system need to work with, a singleton can come in handy to avoid conflicts.

In an object-oriented language, singletons also permit lazy allocation and initialization so that modules that are never used don't consume resources.

Sharing private globals

Even in a procedural language like C, the concept of the singleton has a place. The data-hiding goodness of object-oriented design was noted in "Object-Oriented Programming in C". Protecting a module's variables from modification (and use) by other files will give you a more robust solution.

However, sometimes you need a back door to the information, either because you need to reuse the RAM for another purpose (Chapter 8) or because you need to test the module from an outside agency (Chapter 3). In C++, you might use a friend class to gain access to otherwise hidden internals.

In C, instead of making the module variables truly global by removing the `static` keyword, you can cheat a little and return a pointer to the private variables:

```
static struct sLogStruct gLogData;
struct sLogStruct* LogInternalState() {
  return &gLogData;
}
```

This is not a good way to retain encapsulation and keep data hidden, so use it sparingly. Consider guarding against it during normal development:

```
static struct sLogStruct gLogData;
struct sLogStruct* LogInternalState() {
#if PRODUCTION
  #error "Internal state of logging protected!"
#else
  return &gLogData;
#endif /* PRODUCTION */
}
```

As you design your interface and consider the state information that will be part of your modules, keep in mind that you will also need methods for verifying your system.

A Sandbox to Play In

That pretty much covers the low- and mid-level boxes, but we still have some algorithm boxes to think about. One goal of a good architecture is to keep the algorithm as segregated as possible. The common pattern of *Model-View-Controller* (MVC) is an excellent one to apply here. The purpose of the pattern is to isolate the gooey center of the application from the user interface so they can be developed and tested independently.

Usually MVC is used to allow different displays, interfaces, and platform implementations. However, it can also be used to allow the application and algorithms to be developed separately from the hardware. In this pattern, the view is the interface to the user, both input and output. In our device, the user might not be a person; it could be hardware sensors (input) and a screen (output). In fact, if you have a system without a screen but that sends data over a network, the view may have no visual aspect, but it is still a part of the system as the form of input and output. The model is the domain-specific data and logic. This is the part that takes raw data from the input and creates something useful, often using the algorithm that makes your product special. The controller is the glue that works between the model and the view: it handles how to get the input to the model for processing and the data from the model for display or outside communication. There are standard ways for these three elements to interact, but for now it is enough to understand the separation of functions.

DIFFERENT ASPECTS OF THE MODEL-VIEW-CONTROLLER

The good news about the MVC is that nearly everyone agrees that there are three parts to it. The bad news is that this might be the only thing everybody agrees on.

The model holds the data, state, and application logic. If you are building a weather station, the model has the temperature monitoring code and predictive models. For an MP3 player, the model consists of a database of music and the codec necessary to play it.

Traditionally thought of in contexts where there is a screen, the view represents the display-handling functions. The view is what the user sees of the model. A view could be a picture of a sun or a detailed readout of the statistics from a weather service. Those are both views of the same information.

The controller is the somewhat nebulous cloud that sits between (or near) the model and view to help them work together. The goal of the controller is to make the view and model independent of each other so that each can be reused. For that MP3 player, your company may want a consistent interface even when your audio-playing hardware is revamped. Or maybe it is the same hardware but marketing wants to make the system look more child-friendly. How can you achieve both of these, while touching the smallest amount of code? The controller enables the model/view separation by providing services such as translating a user button press into an event for the model.

Figure 2-8 shows a basic interpretation of the MVC and some common variants. There are a lot of different ways to put it together, some of them seemingly contradictory. The term “MVC” is kind of like the adjective “sanguine,” which can mean either murderous or cheerfully optimistic. You may need some context clues to know which it is, but either way, you are probably talking about someone’s mood.

There is another way to use the MVC pattern, which serves as a valuable way to develop and verify algorithms for embedded systems: use a *virtual box*, or *sandbox*. The more complex the algorithm, the more this is a necessity.

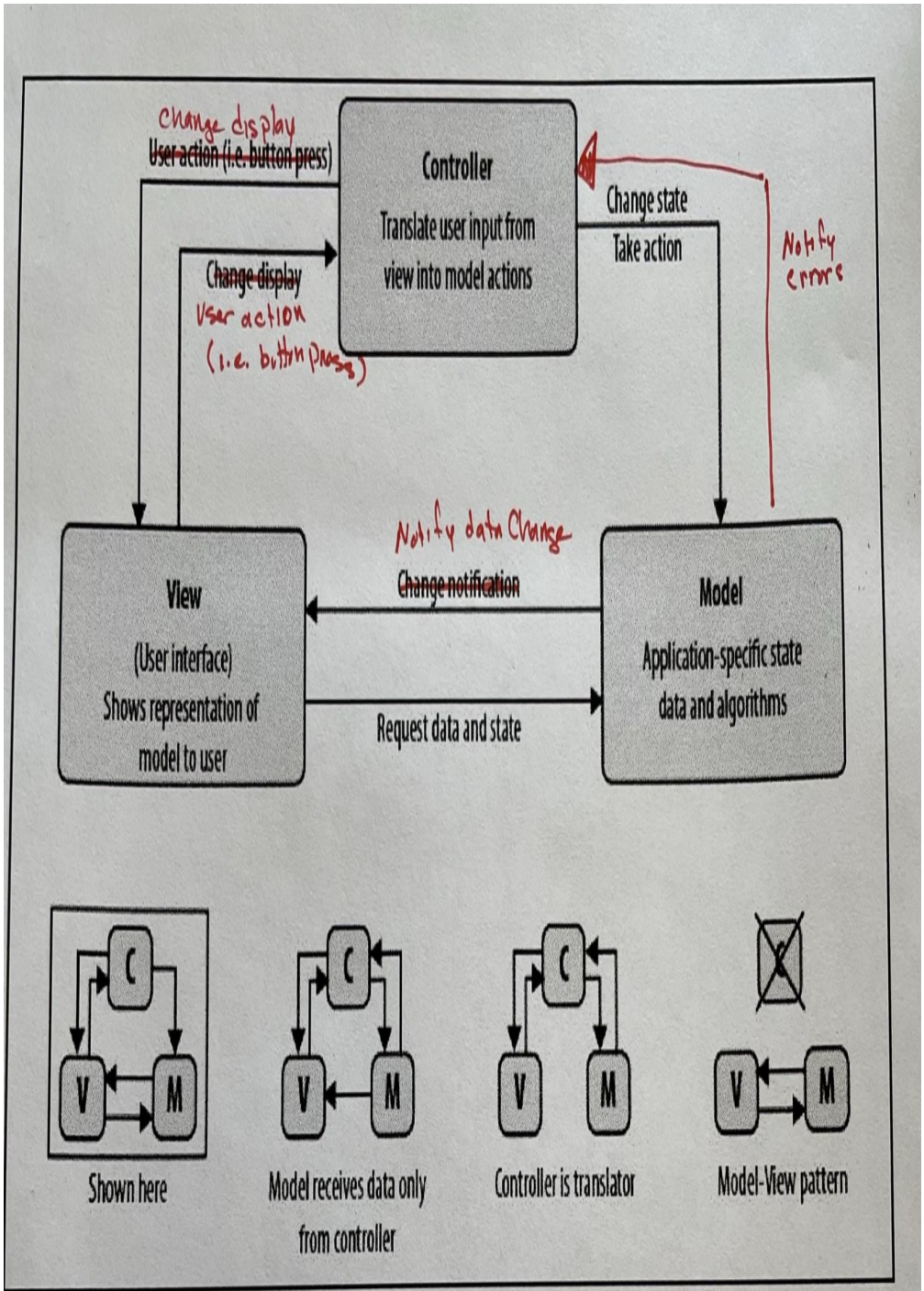


Fig 2-8: overview of Model-View-Controller

fuss/print/mes0208.pdf

Figure 2-9. Overview of Model-View-Controller pattern

If you can make all of the inputs to an algorithm's box and put them through a single interface, you can switch back and forth between a file on a PC and the actual system. Similarly, redirect the output of the algorithm to a file. Now you can test the algorithm on a PC (where the debugging environment may be a lot better than an embedded system), and rerun the same data over and over again until any anomalous behavior can be isolated and fixed. It is a great way to do regression tests to verify that minor algorithm changes don't have unforeseen side effects.

In the sandbox environment, your files and the way you read the data in are the view part of the MVC. The algorithm you are testing is the model, and this shouldn't change. The controller is the part that changes when you put the algorithm on a PC. Consider an MP3 player (Figure 2-10) and what the MVC diagram might look like with a sandbox. Note that both the view and controller will have relatively strict APIs because you will be replacing those as you move from virtual device to actual device.

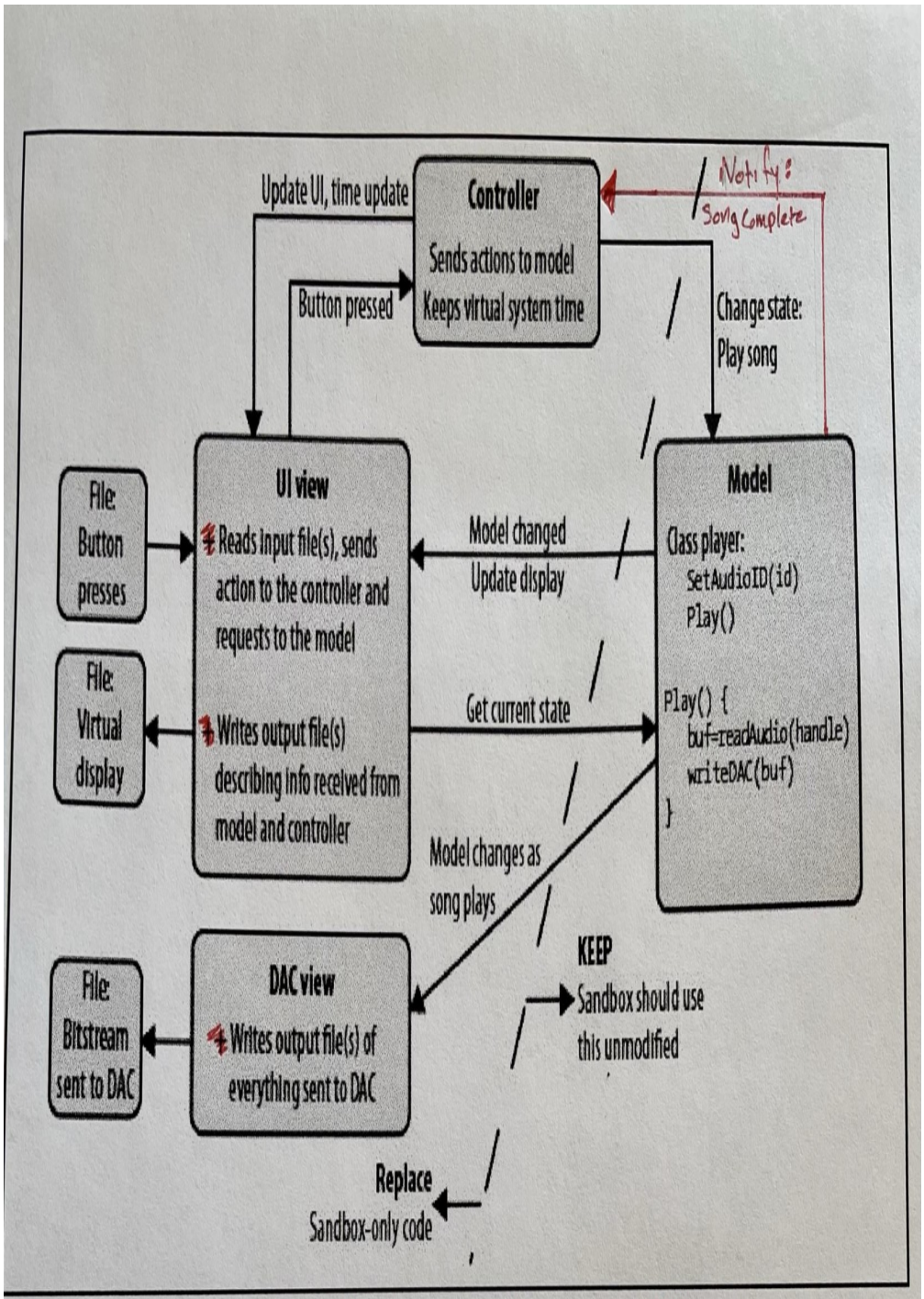


Fig 2-9 Model View Controller in a Sandbox

figs/Print/mvc_0209.pdf

add Notify arrow, remove + signs

Figure 2-10. Model-View-Controller in a sandbox

Your input view file might be like a movie script, directing the sandbox to act as the user might. The first field is a time when the second field, a method in your system, is fired:

```
Time, Action, Variable arguments // comment
00:00.00, PowerOnClean          // do all of the initialization
00:01.00, PlayPressed           // no action, no song loaded
00:02.00, Search, "Still Alive" // expect list back based on available matches
00:02.50, PlayPressed           // expect song to be played
```

The output file can look however you want. For instance, you might have multiple output files, one to describe the state and the changes sent to the user interface from the controller and the model and another one to output what the model would send to the digital-to-analog converter (DAC). The first output file might look like this:

```
Time, Subsystem, Log message
00:00.01, System, Initialization: sandbox version 1.3.45
00:01.02, Controller, Minor error: no song loaded
00:02.02, Controller, 4 songs found for "Still Alive" search, selecting ID 0x1234
00:02.51, Controller, Loading player class to play ID 0x1234
```

Again, you get to define the format so that it meets your needs (which will probably change). For instance, if you have a bug where the player plays the wrong song, the sandbox could output the song ID in every line.

The Model-View-Controller here is a very high-level pattern, specifically a system-level pattern. Once you look at breaking up the system this way, you might find that it has fractal qualities. For instance, what if you only want to be able to test the code that reads a file from the memory and outputs it to a DAC? The filename and the resulting bitstream of information that would go to the DAC could be considered the view (which consists of inputs and outputs, even if no humans are involved), the logic and data necessary to convert the file would be the model, and the file handler may be the controller since that would have to change between platforms.

With this idea of separation and sandboxing in mind, take a look at the architecture drawings and see how many inputs the algorithm has. If there is more than one, does it make sense to have a special interface object? Sometimes it is better to have multiple files to represent multiple things going on. Looking further into the diagrams, does it make sense to incorporate the virtual box at a lower level than just the product feature algorithms? This expands your model and, more importantly, lets you test more of your code in a controlled environment.

Your virtual box could be expensive to develop, so you may not want to start implementation immediately. In particular, your virtual box may have different sizes for your variables and its compiler potentially could act differently, so you may want to hold off building it unless your algorithms are complex or your hardware is a long time in coming. There is a trade-off between time to develop the virtual box and its extraordinary powers of debuggability and testability. But identifying how to get to a sandbox from the design and keeping it in mind as you develop your architecture will give you leeway for when things go wrong.

Back to the Drawing Board

I asked you to make four sketches of your system. If you look at those again with your mind full of encapsulation, data hiding, interfaces, and design patterns, what do you see? The drawings I've chosen are intended to give you different perspectives about the system. Together they will ask questions about the architecture so you think about issues earlier in the process, during design instead of development.

No design will remain static. The goal of design is to understand things from the system perspective. This doesn't mean you need to

update your diagrams with every change. Some of the diagrams may become part of documentation and need to be maintained but these sketches are for you, to improve your understanding of the system and ability to communicate with your team.

The context diagram is a you–are-here map that keeps you on track for what you are supposed to be doing. This is likely to be useful when talking about what the system requirements are.

The block diagram shows the pieces of the hardware and software. The available resources are shown along with a box-sized description of how they are intended to be used. The whole-picture view helps with identifying the tasks that will need to be accomplished. I recommend starting out with non-specific component boxes; the chips these represent will change so how can you harden your code against those changes? The block diagram is the most common drawing and will likely be very useful when talking to colleagues.

The organigram shows where control flows and where there will be conflicts as multiple subsystems try to access the same resources. It can help with

encapsulating modules that don't need to talk to each other (or that do need to talk to each other but shouldn't do so directly). This one may be hard to get your head around but I find the manager relationship helps me think about it: who is bossing this component around? This diagram is most useful for understanding the details of the system and debugging weird errors.

The layering diagram shows how modules can be grouped to provide interfaces and abstraction layers. It helps with identifying what information can be hidden. It can also identify when modules are too big and need to be broken apart. This view can help you reduce complexity and reuse code (or understand those deep vendor-provided hardware abstraction layers),

Each of these diagrams were presented at the whole-system level. However, systems are made of subsystems which are made of subsystems. Making the high level diagram too detailed is a good way to lose the whole-system picture. Sometimes a large box needs to be moved to a new page for more detail (and more thought as you design and understand this subsystem).

If diagramming is useful but my diagrams don't work for you, look up the *C4 model* for visualizing software architecture or the *m4+1 architectural view model*, both are excellent alternatives. For a drawing program, when I'm ready to formalize my sketches, I like *mermaid.live* as it is based on written text and can be put into most markdown documents.

Further Reading

This chapter touched on a few of the many design patterns. The rest of the book will as well, but this is a book about embedded systems, not about design patterns. Think about exploring one of these to learn more about standard software patterns:

- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.

  This is the original, seminal work on design patterns. It uses C++ as the reference language.

- Freeman, Eric T., Elizabeth Robson, Bert Bates, and Kathy Sierra. 2004. *Head First Design Patterns*. Cambridge, MA:

O’Reilly.

Using Java as the example language, this book gives great examples with an engaging style.

- Search on Wikipedia for *software design patterns*.

While we’ll be looking more at developing a module later, if you want to jump ahead or are interested in going beyond the short logging code I put here, take a look at *Embedded Artistry’s Arduino Logging Library [https://embeddedartistry.com/arduino-logger/]*. The author goes through the process of development and ends up with code you can use. (While you are there, click on the *Welcome* link because *Embedded Artistry* has a wealth of code and information.)

INTERVIEW QUESTION: CREATE AN ARCHITECTURE

Describe the architecture for this [pick an object in the room] conference phone.

Looking around an interviewing area is usually fairly dicey because it is devoid of most interesting things. The conference phone gets picked a lot because it is sometimes the most complicated system in the room. Another good target is the projector.

When asking this question, I want to know that the candidate can break a problem down into pieces. I want to see the process as she mentally deconstructs an object. In general, starting with the inputs and outputs is a safe bet. For a conference phone, the speaker and the display are outputs; the buttons and the microphone are inputs. I like to see these as boxes on a piece of paper. A candidate shouldn’t be afraid to pick up the object and see the connections it has. Those are inputs and outputs as well. Once she has the physical hardware down, she can start making connections by asking (herself) how each component works: how does the power button work, and how might the software interface to it? How does the microphone work, and what does it imply about other pieces of the system (e.g., is there an analog-to-digital converter)?

A candidate gets points for mentioning good software design practices. Calling the boxes drivers for the lowest level and objects for the next level is a good start. It is also good to hear some noises about parts of

the system that might be reused in a future phone and how to keep them encapsulated.

I expect her to ask questions about specific features or possible design goals (cost). However, she gets a lot of latitude to make up the details. Want to talk about networking? Pretend it is a voice over IP (VoIP) phone. Want to skip that entirely? Focus instead on how it might store numbers in a mini-database or linked list. I'm happy when she talks about things she is interested in, particularly areas that I've failed to ask about in other parts of the interview.

In that same vein, although I want to see a good overview on the whole system, I don't mind if a candidate chooses to dig deep into one area. This question gives a lot of freedom to expound on how her particular experience would help her design a phone. When I ask a question, I don't mind if she admits some ignorance and talks about something she knows better.

Asking an interviewee about architecture is not about getting perfect technical details. It is crucial to draw something, even if it is only mildly legible. The intent of the question is about the candidate showing enthusiasm for problem solving and effectively communicating her ideas.

[1] Style is very important. Coding guidelines will save you debugging time, and they won't quash your creativity. If you don't have one, look at the style guide Google suggests for open source projects (search "Google Style Guides"). The explanations of why they chose what they did might help you formulate your own guide.

Chapter 3. Getting Your Hands on the Hardware

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at s*evans@oreilly.com*.

It can be tough to start working with embedded systems. Most software engineers need a crash course in electrical engineering, and most electrical engineers need a crash course in software design. Working closely with a member of the opposite discipline will make getting into embedded systems much easier. Some of my best friends are electrical engineers.

If you've never been through a development cycle that includes hardware, it can be a bit daunting to try to intuit your roles and responsibilities. In this chapter, I'll start out with an overview of how a project usually flows. Then I'll give you some more detailed advice on the skills you need to hold up your end of the team, including:

- Reading a datasheet
- Getting to know a new processor
- Unraveling a schematic
- Creating your own debugging toolbox
- Testing the hardware (and software)

Hardware/Software Integration

The product starts out as an idea or a need to be filled. Someone makes a high-level product design based on features, cost, and time to market. At this point, a schedule is usually created to show the major milestones and activities (see Figure 3-1).

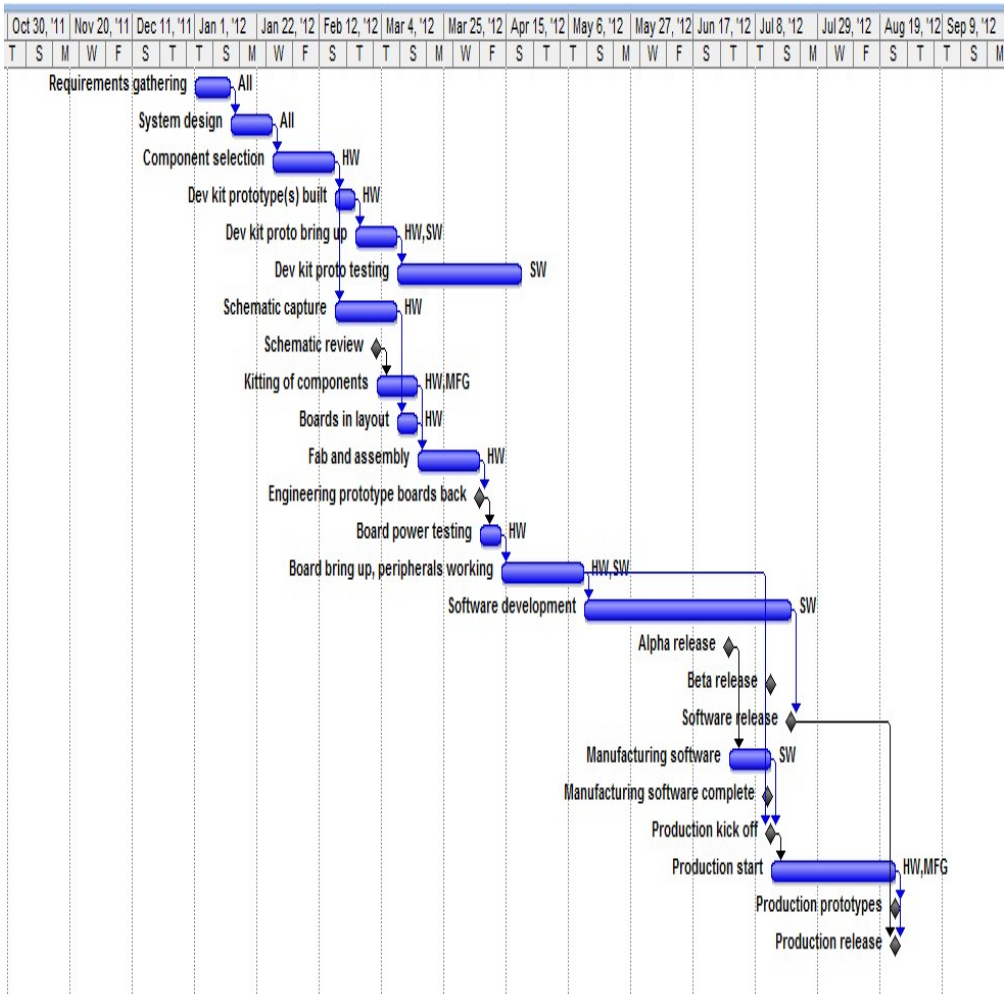


Figure 3-1. Ideal project schedule example

Ideal Project Flow

This sort of schedule is based on the engineering managers (or tech leads) talking about the goals then estimating the time and resources it will take. This is only part of the schedule, it doesn't include the mechanical engineering, server-side software, marketing launch, or a host of other activities. On the other hand, this is likely the part of the schedule that

includes hardware and embedded software so it likely shows your work and that of your immediate team.

The schedule in Figure 3-1 shows a waterfall type of development, where one activity flows into the next. It makes sense that a schematic review is necessary before the boards get created. Of course the boards are created before embedded software development has an alpha release.

But this won't be the actual schedule. It can be quite frustrating to spend a lot of time creating such a masterpiece only to have it be out-of-date before it is finished. The goal of planning is to identify dependencies. If it takes 12 weeks to make the boards and 14 weeks to create the embedded software, then the whole project can be done in 14 weeks, right? Well, no, the software likely needs more time with the completed boards. But hopefully it doesn't need to be 26 weeks (12+14).

A large part of putting together a schedule is figuring out which tasks can be done in parallel versus those that must be done after hardware is complete. Too often the focus is on the end date, the all-important ship date, but there is more information in the schedule: dependencies, human resource needs, and when cash needs to be spent (aside from salaries).

Of course, no matter how much scheduling and planning you do, nothing ever goes according to plan. There will be overruns and backtracking. The act of planning helps you figure out where you need to be careful and which dependencies matter in the long run (and which don't).

Sadly, embedded software is often at the end, trying to make up for delays due to changing requirements or hardware fixes. You develop with partial hardware, with prototype boards and will make good bring up tests so you'll be ready when the boards finally arrive.

You may develop using agile methods instead of waterfall, allowing for more and faster releases. But at some level, creating physical, shippable products, you will likely need to look into the pieces your firmware depends upon and how they get developed.

Hardware Design

Once they know what they are building, the hardware team goes through datasheets and reference designs to choose components, ideally consulting the embedded software team. Often development kits are purchased for the

riskiest parts of the system, usually the processor and the least-understood peripherals (more on processors and peripherals in a bit).

The hardware team creates schematics while the software team works on the development boards. It may seem like the hardware team is taking weeks (or months) to make a drawing, but most of that time is spent wading through datasheets to find a component that does what it needs to for the product, at the right price, in the correct physical dimensions, with a good temperature range, and so on. During that time, the embedded software team is finding (or building) a tool chain with compiler and debugger, creating a debugging subsystem, trying out a few peripherals, and possibly building a sandbox for testing algorithms.

Schematics are entered using a schematic capture program (aka CAD package). These programs are often expensive to license and cumbersome to use, so the hardware engineer will usually generate a PDF schematic at checkpoints or reviews for your use. Although you should take the time to review the whole schematic (described in “Reading a Schematic”), the part that has the largest impact on you is the processor and its view of the system. Because hardware engineers understand that, most will generate an I/O map that describes the connection attached to each pin of the processor (Chapter 4 suggests taking the I/O map and making a header file from it).

Once the schematic is complete (and the development kits have shown that the processor and risky peripherals are probably suitable), the board can be laid out. In layout, the connections on the schematic are made into a drawing of physical traces on a board, connecting each component as they are in the schematic.

NOTE

Board layout is often a specialized skill, different from electrical engineering, so don’t be surprised if someone other than your hardware engineer does the board layout.

After layout, the board goes to fabrication (fab), where the printed circuit boards (PCBs) are built. Then they get assembled with all of their components. Gathering the items on the BOM so you can start assembly is called putting together the *kits*. Getting the long lead time parts can make it

difficult to create a complete kit, which delays assembly. An assembled board is called a PCBA (or PCA), which is what will sit on your desk.

When the schematic is done and the layout has started, the primary task for the embedded software team is to define the hardware tests and get them written while the boards are being created. Working on the user-facing code may seem more productive, but when you get the boards back from fabrication, you won't be able to make progress on the software until you bring them up. The hardware tests will not only make the bring-up smoother, but they also will make the system more stable for development (and production). As you work on the hardware tests, ask your electrical engineer which parts are the riskiest. Prioritize development for the tests for those subsystems (and try them out on a development kit, if you can).

When the boards come back, the hardware engineer will power them on to verify there aren't power issues, possibly verifying other purely hardware subsystems. Then you get a board (finally!) and bring-up starts.

Board Bring-Up

This can be a fun time, one where you are learning from an engineer whose skill set is different from yours. Or it can be a shout-fest where each team accuses the other of incompetence and maliciousness. Your call.

The board you receive may or may not be full of bugs. Electrical engineers don't have an opportunity to compile their code, try it out, make a change, and recompile. Making a board is not a process with lots of iterations. Remember that people make mistakes, and the more fluffheaded the mistake, the more likely it will take an eternity to find and fix (what was the longest time you ever spent debugging an error that turned out to be a typo?).

Don't be afraid to ask questions, though, or to ask for help finding your problem (yes, phrase it that way). An electrical engineer sitting with you might be what you need in order to see what is going wrong.

Finding a hardware (or software) bug at an early stage of design is like getting a gift. The fewer people who know about your bugs, the happier you'll be. Your product team's bugs reflect upon you, so be willing to give your time and coding skills to unravel a problem. It isn't just polite and professional; it is a necessary part of being on a team. If there is a geographic or organizational separation and getting the hardware engineer's help

requires leaping through hoops, consider some back-channel communications (and trade in lunches). It might make you poor in the short term, but you'll be a hero for getting things done, and your long-term outlook will be good.

NOTE

Don't be embarrassed when someone points out a bug; be grateful. If that person is on your team or in your company, that bug didn't get out to the field where it could be seen by customers.

To make bring-up easier on both yourself and your hardware engineer, first make each component individually testable. If you don't have enough code space, make another software subproject for the hardware test code (but share the underlying modules as much as possible).

Second, when you get the PCBA, start on the lowest-level pieces, at the smallest steps possible. For example, don't try out your fancy motor controller software. Instead, set an I/O device to go on for a short time, and hook up an LED to make sure it flashes. Then (still taking things one step at a time), do the minimum necessary to make sure the motor moves (or at least twitches).

Finally, make your tools independent of *you* being present to run them, so that someone else is able to reproduce an issue. Once an issue is reproducible, even if the cause isn't certain, fixes can be tried. (Sometimes the fix will explain the cause.) The time spent writing good tests will pay dividends. There is a good chance that this hardware verification code will be around for a while, and you'll probably want it again when the next set of boards comes back. Also, these tests tend to fold into manufacturing tests used to check the board's functionality during production.

How do you know what tests need to be written? It starts with an in-depth knowledge of the processor and peripherals. That leads to the topic of reading a datasheet.

Reading a Datasheet

With the pressure to get products released, it can be tough to slow down enough to read the component datasheets, manuals, and application notes. Worse, it can feel like you've read them (because you've turned all the

pages), but nothing sticks, because it is a foreign language. When the code doesn't work, you're tempted to complain that the hardware is broken.

If you are a software engineer, consider each chip as a separate software library. Think of all the effort you would need to put into learning a software package (Qt, OpenCV, Zephyr, Unity, TensorFlow, etc.). That is about how long it will take to become familiar with your processor and its peripherals. The processors will even have methods of talking to the peripherals that are somewhat like APIs. As with libraries, you can often get away with reading only a subset of the documentation—but you are better off knowing which subset is the most important one for you before you end up down a rabbit hole.

Datasheets are the API manuals for peripherals. Make sure you have the latest version of your datasheet (check the manufacturer's site and do an online search just to be sure) before I let you in on the secrets of what to read twice and what to ignore until you need it.

WARNING

Unfortunately, some manufacturers release their datasheets only under NDA, so their websites contain only an overview or summary of their product.

Hardware components are described by their datasheets. Datasheets can be intimidating, as they are packed densely with information. Reading datasheets is an art, one that requires experience and patience. The first thing to know about datasheets is that they aren't really written for you. They are written for an electrical engineer, more precisely for an electrical engineer who is already familiar with the component or its close cousin.

In some ways, reading a datasheet is like coming into the middle of a technical conversation. Take a look at Figure 3-2, which resembles a real datasheet, even though it's not a component you're likely to interface with in your software (it is a prehistoric gait controller for leg actuators).

Near the top of each datasheet is an overview or feature list. This is the summary of the chip, but if you haven't already used a component with a datasheet that is 85% the same as this one, the overview isn't likely to be very helpful. On the other hand, once you have used three or four analog

triceratops (or accelerometers, or whatever you are really looking at), you can just read the overview of the next one and get an idea of how this chip is the same and different from your previous experience. If the datasheet took the time to explain everything a newcomer might want to know, each one would be a book (and most would have the same information). Additionally, it isn't the newcomers who buy components in volume; it is the experienced engineers who are already familiar with the devices.

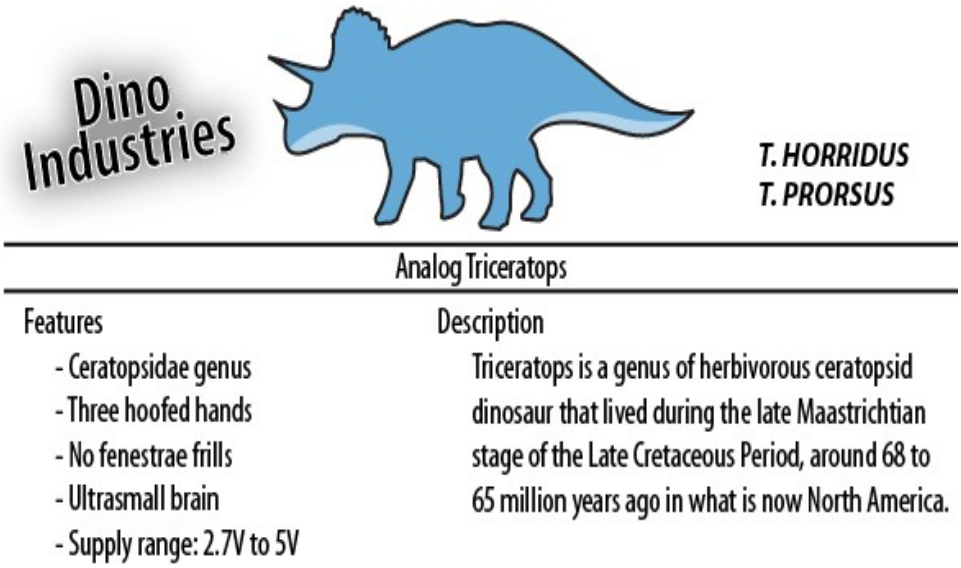


Figure 3-2. Top of the Analog Triceratops datasheet from Dino Industries

So I say skip the overview header (or at least come back to it later).

I like the functional diagrams that are often on the first page, but they are a lot like the overview. If you don't know what you are looking for, the block diagram probably isn't going to enlighten you. So the place to start is the description. This usually covers about half the first page, maybe extending a little on to the second page.

The description is not the place to read a few words and skip to the next paragraph. The text is dense and probably less than a page long. Read this section thoroughly. Read it aloud, underline the important information (which could be most of the information), or try to explain it to someone else (I keep a teddy bear in my office for this purpose, but some people use their EEs).

Datasheet Sections You Need When Things Go Wrong

After the first page, the order of information in each datasheet varies. And not all datasheets have all sections. Nonetheless, you don't want to scrutinize sections that won't offer anything that helps your software use the peripheral. Save some time and mental fatigue by skipping over the absolute maximum ratings, recommended operating conditions, electrical characteristics, packaging information, layout, and mechanical considerations.

The hardware team has already looked at that part of the sheet; trust them to do their jobs.

Although you can skip the following sections now, remember that these sections exist. You may need them when things go wrong, when you have to pull out an oscilloscope and figure out why the driver doesn't work, or (worse) when the part seems to be working but not as advertised. Note that these sections exist, but you can come back to them when you need them; they are safe to ignore on the first pass.

Pin out for each type of package available

You'll need to know the pin out if you have to probe the chip during debugging. Ideally, this won't be important, because your software will work and you won't need to probe the chip.

The pin configuration shows some pin names with bars over them (Figure 3-3). The bars indicate that these pins are *active low* meaning that they are on when the voltage is low. In the pin description, active low pins have a slash before the name. You may also see other things to indicate active low signals. If most of your pins have a `NAME`, look for a modifier that puts the name in the format `nNAME`, `_NAME`, `NAME*`, `NAME_N`, etc. Basically, if it has a modifier near the name, look to see whether it is active low, which should be noted in the pin descriptions section.

If this is all brand new to you, consider picking up a beginner's guide to electronics, such as one of those listed in "Further Reading".

Pin descriptions

This is a table that looks like Figure 3-4. Come back when you need this information, possibly as you look to see whether the lines should be active low or as you are trying to make your oscilloscope show the same image as the timing diagrams in the datasheet.

Performance characteristics

These tables and graphs, describing exactly what the component does, offer too much detailed information for your first reading. However, when your component communicates but doesn't work, the performance characteristics can help you determine whether there might be a reason (e.g., the part is rated to work over 0°C–70°C but it is right next to your extremely warm processor, or the peripheral works when the input voltage is just right but falls off in accuracy when the input gets a little outside the specified range).

Sample schematics

Sometimes you get driver code, it works as you need it, and you don't end up changing much, if anything. The sample schematics are the electrical engineering version of driver code. When your part is acting up, it can be reassuring to see that the sample schematics are similar to your schematics. However, as with vendor-provided driver code, there are lots of excellent reasons why the actual implementation doesn't match the sample implementation. If you are having trouble with a part and the schematic doesn't match, ask your electrical engineer about the differences. It may be nothing, but it could be important.

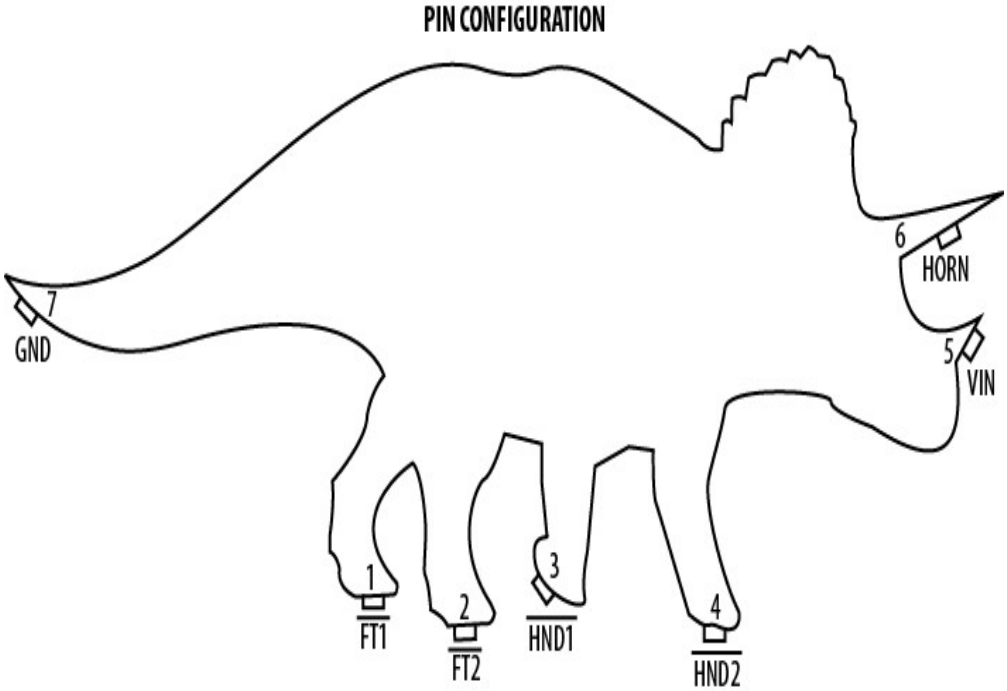


Figure 3-3. Analog Triceratops pin out

Symbol name	Pin	Type	Description
/FT1	1	O	Four-hoofed foot channel 1
/FT2	2	O	Four-hoofed foot channel 2
/HND1	3	O	Three-hoofed hand channel 1
/HND2	4	O	Three-hoofed hand channel 2
VIN	5	I	Power supply
HORN	6	I	Mode setting
GND	7	I	Ground

Figure 3-4. Analog Triceratops pin descriptions

Datasheet Sections for Software Developers

Eventually, possibly halfway through the datasheet, you will get to some text (Figure 3-5). This could be titled “Application Information” or “Theory of Operation.” Or possibly the datasheet just switches from tables and graphs to text and diagrams. This is the part you need to start reading. It is fine to start by skimming this to see where the information you need is located, but eventually you will really need to read it from start to end. As a bonus, the text there may link to application notes and user manuals of value. Read through the datasheet, considering how you’ll need to implement the driver for your system. How does it communicate? How does it need to be initialized? What does your software need to do to use the part effectively? Are there strict timing requirements, and how can your processor handle them?

As you read through the datasheet, mark areas that may make an impact on your code so you can return to them. Timing diagrams are good places to stop and catch your breath (Figure 3-6). Try to relate those to the text and to your intended implementation. More information on timing diagrams is provided in the sidebar “How Timing Diagrams Help Software Developers”.

THEORY OF OPERATION

In normal mode, the triceratops has a phase shifted output where the opposite signal is synchronized (FT1 with HND2, FT2 with HND2). Each signal is high for approximately one quarter of the cycle.

In charge mode (HORN pulled low), all output pins are synchronized. When the triceratops is charging, it spends less time with its hands and feet in the ground state: each signal is low for approximately one quarter of each cycle. It also has a higher cycle rate which gets incrementally faster until it reaches its maximum charge speed. It maintains this output until the charge is ended (HORN pulled high) or until the triceratops encounters undue resistance (crashes) and boot mode is reinitiated. While not necessary, the HORN is generally pulled high after the crash to prevent overheating of the component.

Figure 3-5. Analog Triceratops theory of operation

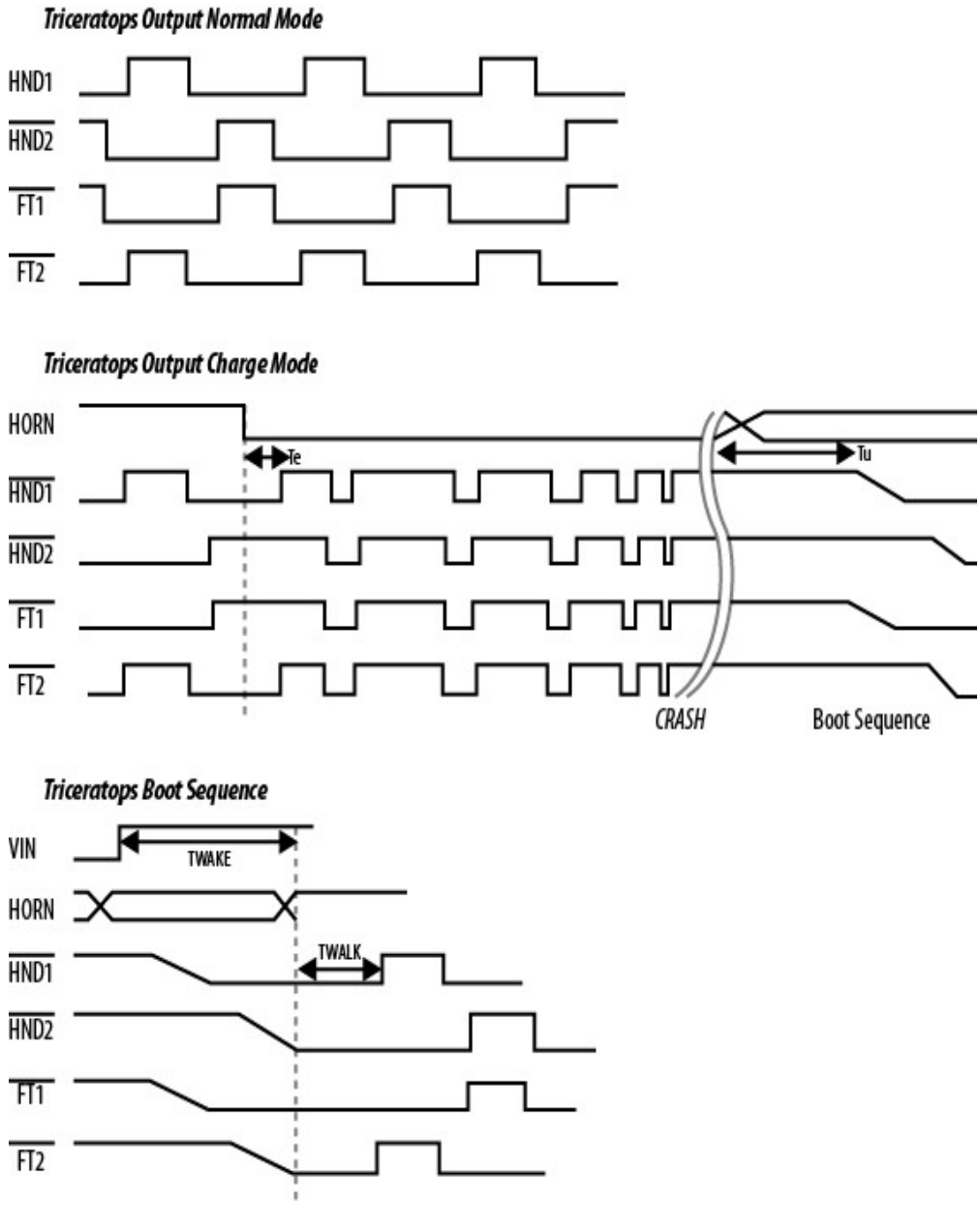


Figure 3-6. Analog Triceratops timing diagrams

In addition to making sure you have the latest datasheet for your component, check the manufacturer's web page for errata, which provides corrections to any errors in the datasheet. Search the web page for the part number and the word "errata" (Figure 3-7). Errata can refer to errors in the datasheet or errors in the part.

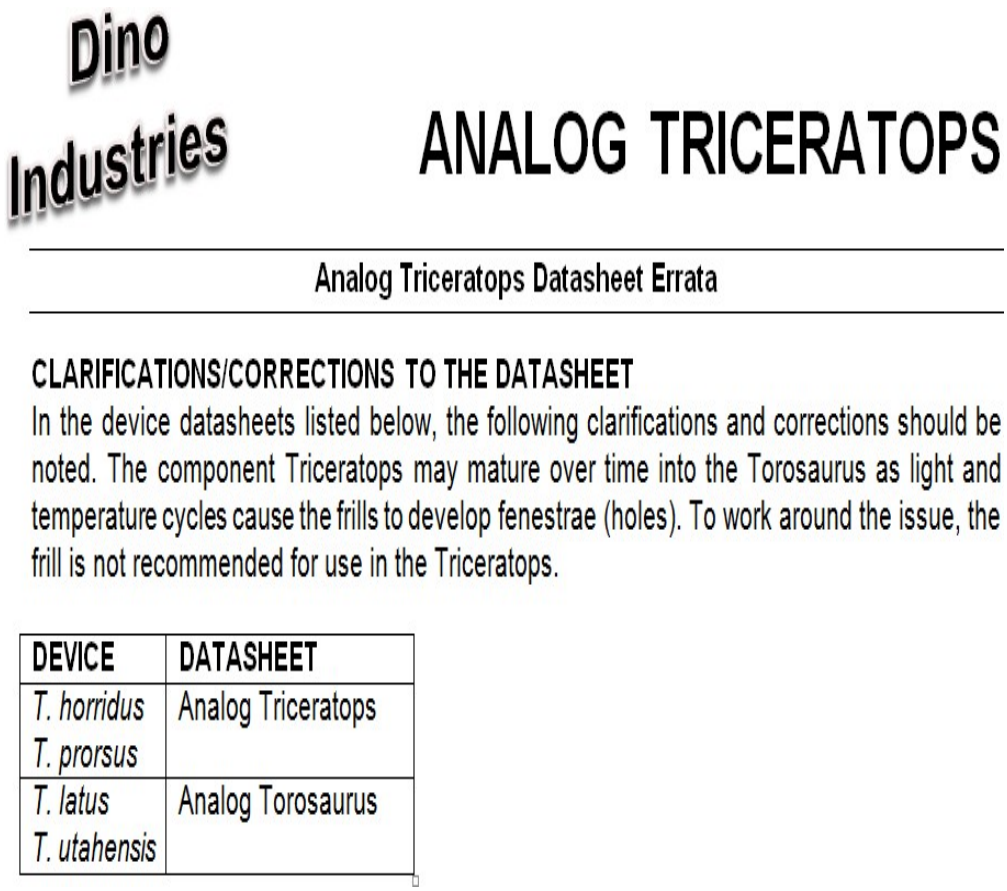


Figure 3-7. Analog Triceratops errata

NOTE

Datasheets can have revisions, and components can have revisions. Get the latest datasheet for the component revision you are using.

When you are done, if you are very good or very lucky, you will have the information you need to use the chip. Generally, most people will start writing the driver for the chip and spend time re-reading parts as they write the interface to the chip, then the communication method, then finally actually use the chip. Reading datasheets is a race where the tortoise definitely wins over the hare.

Once you are all done with the driver and it is working, read through the feature summary at the top of the datasheet because now you have conquered this type of component and can better understand the summary. Even so, when you implement something similar, you'll probably still need to read

parts of the datasheet, but it will be simpler and you'll get a much better overview from the summary on the datasheet's front page.

Other resources may also be available as you work with peripherals:

- If the chip has a user manual, be sure to look at that.
- Application notes often have specific use cases that may be of interest. These often have vendor example code associated with them. Also look for a vendor code repository.
- Search the part on GitHub or the web. Example code from other folks using the part can show you what you should be doing.

Look around before diving in; the answer to something you don't know yet may be out there.

HOW TIMING DIAGRAMS HELP SOFTWARE DEVELOPERS

Timing diagrams show the relationship between transitions. Some transitions may be on the same signal, or the timing diagram may show the relationship between transitions on different signals. For instance, alternating states of the hands (HND1 and HND2) in normal mode at the top of Figure 3-6 show that one hand is raised shortly after the other is put down. When approaching a timing diagram, start on the left side with the signal name. Time advances from left to right.

Most timing diagrams focus on the digital states, showing you when a signal is high or low (remember to check the name for modifiers that indicate a signal is active low). Some diagrams include a ramp (such as hands and feet in the boot sequence in Figure 3-6) to show you when the signal is in a transitory state. You may also see signals that are both high and low (such as the horn in the boot sequence of Figure 3-6); these indicate the signal is in an indeterminate logical state (for output) or isn't monitored (for input).

The important time characteristics are highlighted with lines with arrows. These are usually specified in detail in a table. Also look for an indication of signal ordering (often shown with dashed lines) and causal

relationships (usually arrows from one signal to another). Finally, footnotes in the diagram often contain critical information.

Evaluating Components Using the Datasheet

It may seem odd to have this section about evaluating a component after its implementation. That isn't the way it goes in projects. However, it is the way it goes in engineering life. Generally, before you get to the point where you choose pieces of the system, you have to cut your teeth on implementing pieces of systems designed by other people. So evaluating a component is a more advanced skill than reading the datasheet, a skill that electrical engineers generally develop before software engineers.

When you are evaluating a component, your goal is to quickly eliminate components that won't work. Don't waste valuable time determining precisely how a component does feature X if the component can't be used because it requires 120 V AC and your system has 5 V of DC. Start off with a list of must-haves and a list of wants. Then you'll want to generate your potential pool of parts that require further investigation.

Before you get too deep into that investigation, let's talk about the things that datasheets don't have. They usually don't have prices, because those depend on many factors, especially how many you plan to order. And datasheets don't have lead times, so be careful about designing around the perfect part; it may be available only after a six-month wait. Unless you are ordering online, you'll need to talk to your vendor or distributor.

Talking with your vendor is a good opportunity to ask whether they have any guidance for using the part, initialization code, application notes, white papers, forums, or anything that might get you a little further along. Vendors recognize that these can be selling points, and their application engineers are generally willing to help. Distributors might even help you compare and contrast the different options available to you.

NOTE

Even when working with a distributor, Digikey (http://www.digikey.com/) is often a great way to get some idea of price and lead times.

Back to the datasheet. Some of those information dense sections previously skipped become important. Start with the absolute maximum ratings and electrical characteristics (see Figure 3-8). If they don't match (or exceed) your criteria, set the datasheet aside. The present goal is to wade through the pile of datasheets quickly; if a part doesn't meet your basic criteria, note where it fails and go on. (Keeping notes is useful; otherwise, you may end up rejecting the same datasheet repeatedly.) You may want to prioritize the datasheets by how far out of range they are. If you end up without any datasheets that meet your high standards, you can recheck the closest ones to see whether they can be made to work.

Once the basic electrical and mechanical needs are met, the next step is to consider the typical characteristics, determining whether the part is what you need. I can't help you much with specifics, as they depend on your system's needs and the particular component. Some common questions at this level are about your functional parameters: does the component go fast enough? Does the output meet or exceed that required by your system? In a sensor, is the noise acceptable?

Once you have two or three datasheets that pass the first round of checking, delve deeper into them. If there is an application section, that is a good place to start. Are any of these applications similar to yours? If so, carry on. If not, worry a bit. It is probably all right to be the first to use a peripheral in a particular way, but if the part is directed particularly to underwater sensor networks and you want to use it in your super-smart toaster, you might want to find out why they've defined the application so narrowly. More seriously, there may be a reason why the suggested applications don't cover all uses. For example, chips directed toward automotive use may not be available in small quantities. The goal of the datasheet is to sell things to people already using similar things, so there may be a good reason for a limited scope.

Next, look at the performance characteristics and determine whether they meet your needs. Often you'll find new requirements while reading this section, such as realizing your system should have the temperature response of part A, the supply voltage response of part B, and the noise resistance of part C. Collect these into the criteria, and eliminate the ones that don't meet your needs (but also prioritize the criteria so you don't paint yourself into a corner).

At this point, you should have at least two but no more than four datasheets per part. If you have more than four, ask around to see whether one vendor has a better reputation in your purchasing department, has shorter lead times, or has better prices. You may need to come back to your extras, but select out four that seem good.

If you have eliminated all of your datasheets or you have only one, don't stop there. It doesn't mean that everything is unusable (or that only one is). Instead, it may mean that your criteria are too strict and you'll have to learn more about the options available to you. So choose the two best components, even if one or both fail your most discerning standards.

For each remaining datasheet, you want to figure out the tricky pieces of the implementation and see how well the component will fare in your system. This is where some experience dealing with similar parts is useful. You want to read the datasheet as though you were going to implement the code. In fact, if you don't have experience with something pretty similar, you may want to type up some code to make it real. If you can prototype the parts with actual hardware, great! If not, you can still do a mental prototype, going through the steps of implementation and estimating what will happen.

Even though you will be doing this for two to four parts and probably using only one of them, this will give you a jumpstart on your code when the part is finally chosen.

Such in-depth analysis takes a significant amount of time but reduces the risk of the part not working. How far you take your prototype is up to you (and your schedule). If you still have multiple choices, look at the family of the component. If you run out of something (space, pins, or range), is there a pin-for-pin compatible part in the same family (ideally with the same software interface)? Having headroom can be very handy.

Finally, having selected the component, the feature summary is an exercise in comparative literature. Now that you have become that person who has already read several datasheets for similar components, the overview that starts the datasheet is for you. If you have any remaining datasheets to evaluate, start there. Compare the ones you've done against the new ones to get a quick feel for what each chip does and how different parameters interact (e.g., faster speed likely will be proportional to higher cost).

Absolute Maximum Ratings
over operating free-air temperature range

Parameter	Rating	Units
VIN to GND	-0.3 to +6	V
Output current	100, momentary	mA
Input current	10, continuous	mA
Maximum junction temperature	150	C
Operating temperature range	-40 to+105	C
Storage temperature range	-60 to +150	C
Animation period	68 to 65	Mya
Top charge speed	24	km/h

Figure 3-8. Analog Triceratops absolute maximum ratings

Your Processor Is a Language

The most important component is the one your software runs on: your processor.

Some vendors (ST, TI, Microchip, NXP, and so on) use a common core (i.e. Arm Cortex-M4F). The vendors provide different on-chip peripherals. How your code interfaces with timers or communication methods will change. Sharing a core will make the change less difficult than moving between a tiny 8-bit processor and a mighty 64-bit processor but you should expect differences.

As you get to know a new processor, expect that it will take almost the same level of effort as if you were learning a new programming language. And, like learning a language, if you've already worked with something similar, it will be simpler to learn the new one. As you learn several programming languages or several processors, you will find that learning the new ones becomes easier and easier.

Although this metaphor gives you an idea of the scale of information you'll need to assimilate, the processor itself is really more like a large library with an odd interface. Talking to the hardware is a misnomer. Your software is actually talking to the processor software through special interfaces called *registers*, which I'll cover in more detail in Chapter 4. Registers are like keywords in a language. You generally get to know a few (`if`, `else`, `while`),

and then later you get to know a few more (`enum`, `do`, `sizeof`), and finally you become an expert (`static`, `volatile`, `union`).

The amount of documentation for a processor scales with the processor complexity. Your primary goal is to learn what you need to get things accomplished. With a flood of information available, you'll need to determine which pieces of documentation get your valuable time and attention. Some documents to look for include:

User Manual (or User Guide or Reference Manual) from the processor vendor

Often voluminous, the user manual provides most of what you'll need to know. Reading the introduction will help you get to know the processor's capabilities (e.g., how many timers it has and its basic memory structure).

NOTE

Why get the user manual from the vendor? Take for example the STM32F103 processor, which uses an ARM Cortex-M3 core. You don't want to read the ARM user manual if you are using the F103, because 88% of the information in the ARM manual is extraneous, 10% will be in the STM32F10x manual, and the last few percent you probably won't ever need.

Often these are written for families of processors, so if you want to use an STM32F103, you'll need to get an STM32F10x manual and look for the notes that differentiate the processors. Once you read the introduction, you'll probably want to skip to the parts that will be on your system. Each chapter usually will have a helpful introduction of its own before moving into gritty details.

The user manual will have the information you need to work with the chip, though it may not help you get a system up and working.

Getting Started Guide or User Manual for the development kit

A development kit (dev kit) is often the place to start when working with a new processor. Using a dev kit lets you set up your compiler and debugger with confidence, before custom hardware comes in (and gives you something to compare against if your

hardware doesn’t work immediately). The dev kit is generally a sales tool for the processor, so it isn’t too expensive and tends to have excellent documentation for setting the system up from scratch. The kit recommends compilers, debuggers, and necessary hardware, and even shows you how to connect all the cables. The development kit documentation is intended to be an orientation for programmers, so even if you don’t purchase a kit, the associated documentation may help orient you to the processor’s ecosystem.

Getting Started Guide (slides)

This document describes how to get started with using the processor for both electrical engineers and software developers. Although interesting and fast to read, this slide deck generally won’t answer questions about how to use the processor. It can be helpful when evaluating a processor for use in a project, as it does discuss what the processor is and common applications. Also, it might give you an idea of what dev kits are available.

Wikis and forums

While the main Wikipedia page to your processor probably won’t have enough information to help you get code written, it may give you a high-level overview (though usually the user manual’s introduction is more useful to you). The Wikipedia page may have valuable links to forums and communities using the processor where you can search for problems you might encounter and see how other people solved them.

The vendor may also have wiki pages or forums devoted to the processor. These can be valuable for another perspective on the information in the user manual or getting started guide. They are often easy to search, with links to lots of examples.

Vendor or distributor visits

Sit in on these. They may have little readily pertinent information, but the networking is useful later when you ask for code or support.

Processor datasheet

> The datasheet for your processor is usually more focused on the electrical aspects. Since you'll be writing the software, you want something more software oriented. So for processors, skip the datasheet and go to the user manual (or user guide).

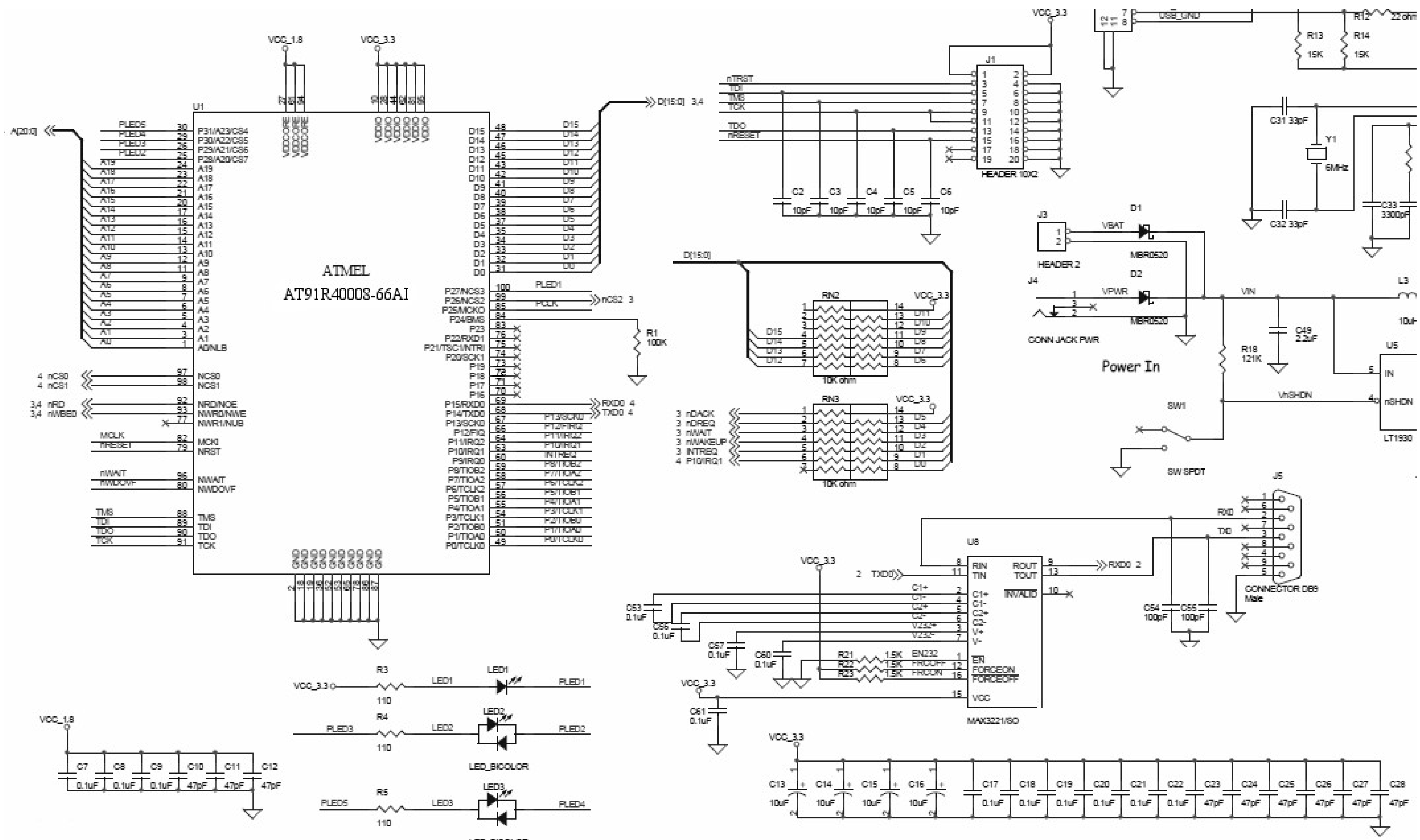
Most processors now come with many examples, including a ton of driver code. Sometimes this is good code, sometimes not. Even with the not-so-great code, it is nice to have an example to start from. Though the examples generally work as described, they probably won't be robust or efficient enough for your needs. If you are going to use the code, it becomes your code to own, so make sure you understand it.

Once you get oriented, preferably with a dev kit up and running, hunker down with the user manual and read the chapters for every interface you are using (even if your vendor gave you example code that does what you need). As we work through the specifics of an embedded system, there will be more details about what to expect from chapters in the user manual (inputs and outputs, interrupts, watchdogs and communications, etc.). For now, let's go back to the bigger picture of the system we are about to bring up.

Reading a Schematic

If you come from the traditional software world, schematics can seem like an eye chart with hieroglyphics, interspersed with strange boxes and tangled lines. As with a datasheet, knowing where to start can be daunting. Beginning on page one of a multipage schematic can be dicey because many electrical engineers put their power-handling hardware there, something you don't necessarily care about. Figure 3-9 shows you a snippet of one page of a schematic.

Some schematics have blocks of textual data (often in the corner of a page). Those are comments; read them carefully, particularly if the block is titled "Processor I/Os" or something equally useful. These are like block comments in code: they aren't always there, so you should take your electrical engineer out to lunch when your schematic is well commented.



ATMEL
AT91R40008-66AI
U1
VCC_1.8
VCC_3.3
A[20:0]
D[15:0] 3,4
nCS2 3
RXD0 4
TXD0 4
R1 100K
J1
HEADER 10X2
nTRST
TDI
TMS
TCK
TDO
nRESET
C2 10pF
C3 10pF
C4 10pF
C5 10pF
C6 10pF
RN2
RN3
10K ohm
J3
HEADER 2
J4
CONN JACK PWR
D1
D2
MBR0520
VBAT
VPWR
VIN
Power In
C49 2.2uF
R18 121K
SW1
SW SPDT
VnSHDN
U5
LT1930
L3
Y1 6MHz
C31 33pF
C32 33pF
C33 3300pF
R12 22 ohm
R13 15K
R14 15K
J5
CONNECTOR DB9 Male
U8
MAX3221/SO
C54 100pF
C55 100pF
R21
R22
R23
1.5K
C53 0.1uF
C56 0.1uF
C57 0.1uF
C60 0.1uF
C61 0.1uF
R3
R4
R5
110
LED1
LED2
LED3
LED_BICOLOR
PLED1
PLED2
PLED3
PLED4
PLED5
C7 0.1uF
C8 0.1uF
C9 0.1uF
C10 47pF
C11 47pF
C12 47pF
C13 10uF
C14 10uF
C15 10uF
C16 10uF
C17 0.1uF
C18 0.1uF
C19 0.1uF
C20 0.1uF
C21 0.1uF
C22 0.1uF
C23 47pF
C24 47pF
C25 47pF
C26 47pF
C27 47pF
C28 47pF

Figure 3-9. Example schematic snippet

WARNING

Most of the time you'll get a hardware block diagram to help you decode a schematic. On the other hand, a question I've been asked in several interviews is, "What does this schematic do?" So this section gives you tips on getting through that as well as a more friendly schematic.

As you go through a schematic for the first time, start by looking for boxes with lots of connections. Often this can be simplified further because box size on the schematic is proportional to the number of wires, so look for the largest boxes. Your processor is likely to be one of those things. Because the processor is the center of the software world, finding it will help you find the peripherals that you most care about.

Above the box is the component ID (U1), often printed on the PCB. Inside or under the boxes is usually the part number. As shown in Figure 3-9, the part number might not be what you are used to seeing. For example, your processor manual may say Atmel AT91R40008, but the schematic may have AT91R40008-66AI, which is more of a mouthful. However, the schematic describes not only how to make the traces on the printed circuit board, but also how to put together the board with the correct components. The extra letters for the processor describe how the processor is packaged and attached to the board.

By now you've found two to four of the largest and/or most connected components. Look at their part numbers to determine what they actually are. Hopefully, you've found your processor. (If not, keep looking.) You may also have found some memory. Memory used to be on the address bus, so it would have almost as many connections as the processor, often with 8 or more address lines and 16 data lines, as in Figure 3-9. Many newer processors have enough embedded memory to alleviate the need for external RAM or flash, so you may not find any large components in addition to your processor.

Next, look at the connectors, which can look like boxes or like long rectangles. These are labeled with Js instead of Us (e.g., J3). There should be a connector for power to the system (at least two pins: power and ground). There is probably a connector for debugging the processor (J1 in Figure 3-9).

The other connectors may tell you quite a bit about the board; they are how the world outside sees the board. Is there a connector with many signals that could indicate a daughter board? Is there a connector with an RS-232 signal to indicate a serial port? A connector filled with wire names that start with USB or LCD? The names are intended to give you a hint.

With connectors and the larger chips identified, you can start building a mental model of the system (or a hardware block diagram if you don't already have one). Now go back to the boxes and see if you can use the names to estimate their function. Looking for names such as SENSOR or ADC might help. Chapter 6 will give you some other signal names that might help you find interesting peripherals.

NOTE

In a schematic, wires can cross without connecting. Look for a dot to indicate the connections. Schematics seldom have a dot at a four-way cross, as it can be difficult to tell if there is a dot there when the schematic is printed.

In all this time, we've been ignoring the non-box-shaped components. Although it is useful to be able to understand a resistor network, an RC filter, or an op-amp circuit, you don't need to know these right away. Figure 3-10 shows some common schematic components and their names, though they will be drawn a little differently in your schematic. This will let you express curiosity about the function of a particular resistor. "Further Reading" gives you some suggestions on how to increase your hardware knowledge.

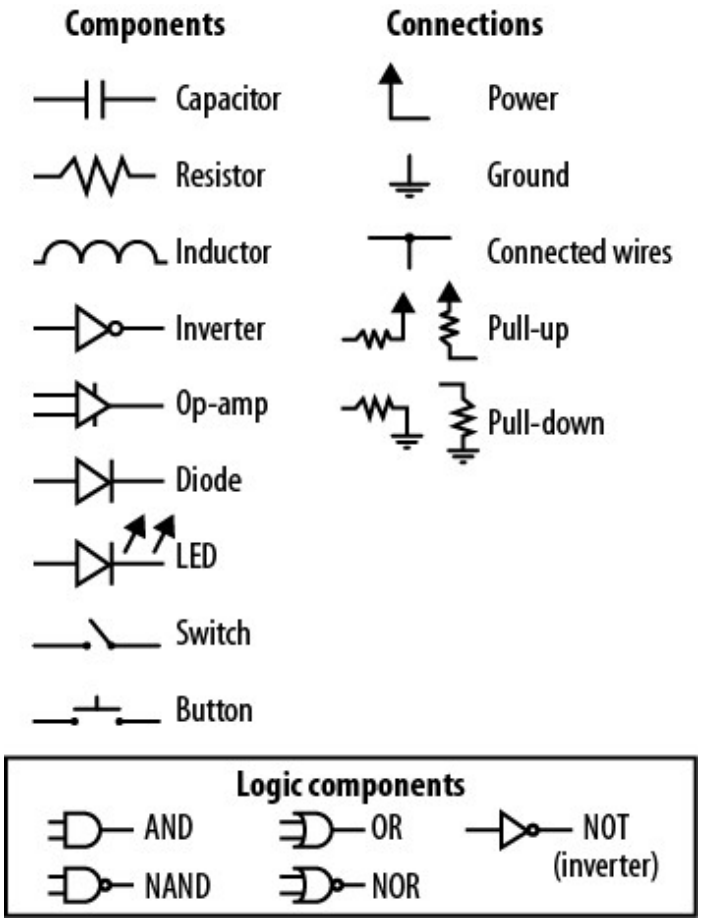


Figure 3-10. Common schematic components

There are two exceptions to my recommendation that you ignore everything but boxes. First, LEDs, switches, and buttons are often connected to the processor. The schematic will tell you the processor line this type of component is connected to so you know where to read the state of a switch or turn on an LED.

The other kind of components to pay attention to are resistors connected to the processor and power, known as pull-ups because they pull the voltage up (toward power). Usually, pull-ups are relatively weak (high amounts of resistance) so the processor can drive a pulled-up I/O line to be low. The pull-up means that the signal on that line is defined to be high when the processor isn't driving it. A processor may have internal pull-ups so that inputs to the processor have a default state even when unconnected.

Note that there are also pull-downs, which means resistors to ground. All of the pull-up information applies to them, except that their default logic level is low instead of high. Inputs without a pull-up or pull-down are driven to

neither a logical high nor low, so they are *floating* (they can also be called hi-Z, high impedance, or tristated).

Practice Reading a Schematic: Arduino!

Arduinos are awesome little boards, designed to be easy to use, to make electronica accessible to artists, students, and generally everyone. For many people, the Arduino is the gateway into embedded systems, it is the first hardware they've ever programmed.

I want to show you how to go from looking at the Arduino as a board that is a complete system, a black box, to looking at it as a board whose schematic is familiar.

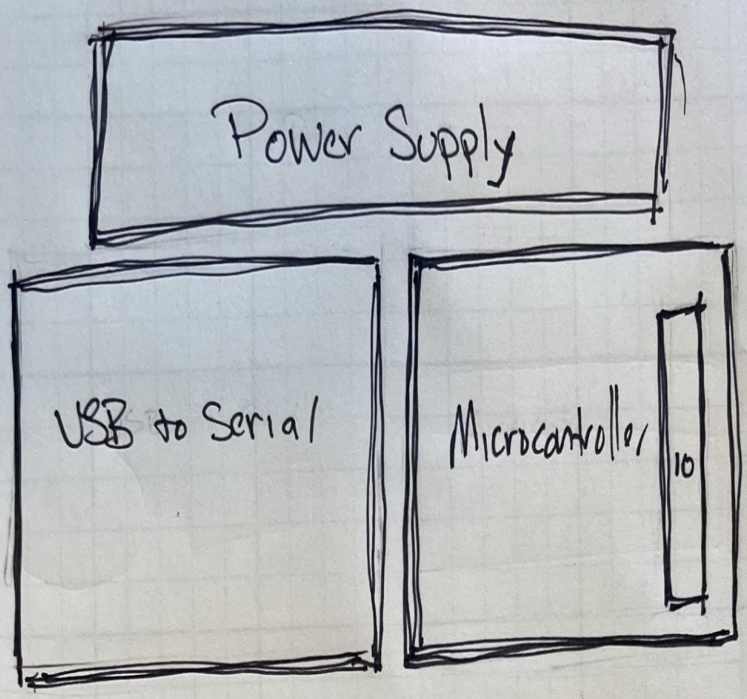


Figure 3-10 #1: Arduino Hardware Block Diagram

NOTE: Please use sketch style from Chapter 2. Also keep the proportions about the same.

DESCRIPTION: The hardware block diagram starts with a power supply box on the top covering most of the horizontal area and about 1/3 of the vertical area. Below that, on the left, is the "USB-to-Serial" box covering the lower 2/3 to near the center. The "Microcontroller" box is to the right covering the rest of the space. Inside the microcontroller box, near the right edge is an IO box.

Figure 3-11. Arduino Hardware Block Diagram

Figure 3-11 shows a hardware block diagram of the Arduino UNO schematic. On the bottom right is the microcontroller. This is the end-user feature of the board. The IO pins come directly from the microcontroller but are kind of separate on the board (and are an important feature to this system) so they get their own box.

To the left of the microcontroller, there is a USB-to-UART (or USB-to-serial) box representing the hardware that translates from what your computer speaks to something the microcontroller understands. It is used for programming the board with new code as well as for printing out data as your code runs. On most block diagrams this piece would be a small box; it is a minor feature, sometimes part of a cable. However, I'm building up to a schematic so I'm keeping it about as large as the space it takes on the schematic.

The final box, on top, is the power subsystem. The Arduino board is a little odd because you can plug it into USB or into a 5V DC wall wart. If you think about the electronic devices around you (a phone, mouse, webcam), it is usually one or the other. But on this board you can plug into both and not blow anything up. The power subsystem is more complicated because of that flexibility.

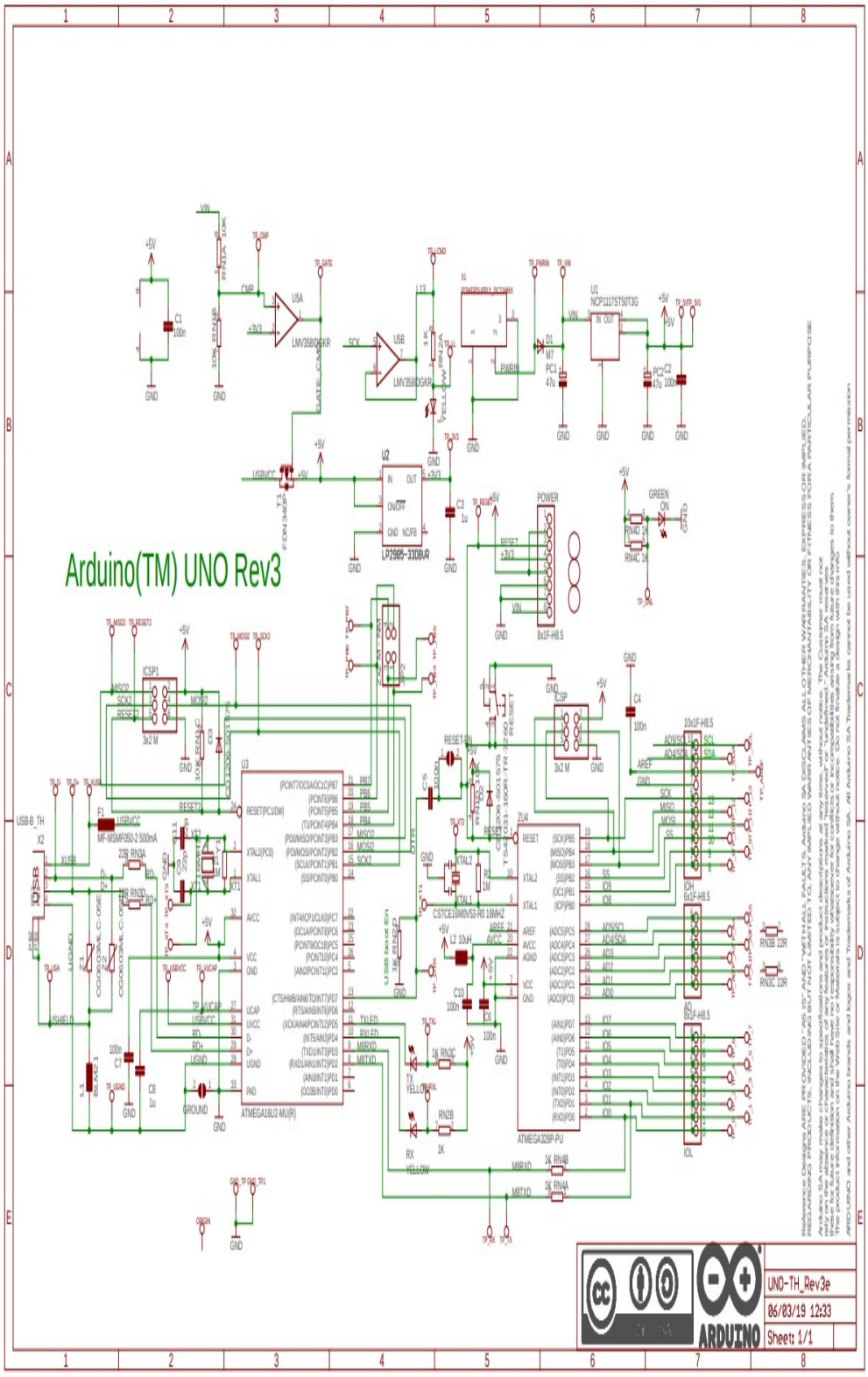


Figure 3-12. Arduino Schematic, retrieved from https://content.arduino.cc/assets/UNO-TH_Rev3e_sch.pdf

Take a look at Figure 3-12. Even better, print out your own copy and get out some colored pencils.

Draw the hardware block diagram over the schematic. The microcontroller is on the bottom right with the IO pins on the very right. Then there are some passive components (which we don't care about so we'll just skip those pesky symbols. On the left of those is the USB-to-UART section (and to the left of that, more pesky passives as well as the USB connector on X2).

Over the top is the power subsystem. Find a connector marked with the word POWER. Everything above that is power related stuff.

What else can you find on the schematic?

- If we go with the "larger things are more interesting" methodology, we'd find the ATMEGA16U2-MU(R) and the ATMEGA328P-PU are the two big chips. From the block diagram I gave you, the ATMEGA328P-PU is the microcontroller for the user. The other is the USB-to-UART section.
- Can you find the LEDs? Not only do the names include the color, look for the triangle symbol with arrows. These are useful on your schematics.
- Look for ICSP and ICSP1 (they are above the chips). These are connectors used to program the boards in manufacturing. If you have an Arduino UNO board, can you find these connectors?Are they different from the connectors?

If you take some time to look for the ATMEGA328 datasheet, you'll find it has two pages of dense, possibly impenetrable, summary information. Then it will show pin descriptions that look similar to the physical processor on your board. The datasheet has a comparison of different, similar processors. It will talk about the core of the processor (the AVRCore). This is an interesting view into microcontroller design but the compiler will hide most of this behind the magic of our programming language. There will be a chapter about memories: Flash, RAM, EEPROM. Then chapters about each subsystem and communication protocol. The datasheet has all the basic information.

If you've been following along, you may have already noticed that Microchip is the vendor of this chip. If you find the processor on the Microchip site, you'll discover pages and pages of documents and application notes that dive deeper (ADC Basics, Reference Manuals, Errata and dozens more). The vendors provide this information because they want to sell their chips.

Keep Your Board Safe

The datasheets, user manuals, and schematics are just paper (or electronic forms of paper). Let's get to the hardware. Wait: before you grab it, be aware that touching hardware can give it an electrostatic zap and damage it (especially if you've just taken off a fleece jacket).

Ask your hardware engineer for the tools to keep your board safe. Try to be conscious of what your board is sitting on. Always carry it around in the bag it came in. Anti-static mats are cheap and force you to allocate a portion of your desk for hardware (even if you don't use the anti-static wrist band, it is still an improvement over crushing the hardware under a falling book or spilling coffee on it).

If possible, when any wires are added to the board, get them glued down as well as soldered. Many an hour has been lost to a broken rework wire. In that same vein, if the connectors are not keyed so that they can be inserted only one way, take a picture of the board or make notes about which way is the correct way to plug in a connector.

I like having my hardware connected to a power strip I can turn off (preferably one my computer is not connected to). It is really good to have an emergency stop-all plan. And note where the fire extinguisher is, just in case.

Seriously, the problem usually isn't flames but more like little puffs of smoke. Whatever you do to it, though, a damaged board is unlikely to win friends. When your manager or hardware engineer asks you how many boards you want allocated for the embedded software, always ask for a spare or two. Nominally this is so you can make sure the hardware tests work on more than one board. Once the system starts making baby steps, you are the person most likely to lose access to a board so it can be shown off to others. At least those are the reasons you should give for wanting a spare board and not because you may very well damage the first one (or two).

NOTE

Often there are also manufacturing errors, especially early on in dense board designs. The bare PCB will be tested for continuity, but there is only so much that can be done, especially with a new design, to test for proper solder connections with no breaks and no shorts. If something doesn't work on one piece of hardware, it is really useful to have a second to try it on, plus a third as a tie-breaker if you get different results on two boards.

Creating Your Own Debugging Toolbox

I love my toolbox because it gives me some level of independence from my hardware engineer. With the toolbox on my desk, I can make small changes to my board in a safer way (using needle-nose pliers to move a jumper is much less likely to damage a board than using fingers). Not counting the detritus of assorted jumper wires, RS-232 gender changers, and half-dead batteries I've accumulated over the years, my toolbox contains:

- Needle-nose pliers
- Tweezers (one pair for use as mini-pliers, one pair for use as tweezers)
- Box cutter
- Digital multimeter (more on this in a minute)
- Electrical tape
- Sharpies
- Assorted screwdrivers (or one with many bits)
- Flashlight
- Magnifying glass
- Safety glasses
- Cable ties (Velcro and zip tie)

If your company has a good lab with someone who keeps the tools in labeled areas, use those (but still have your own). If not, a trip to the hardware store may save some frustration in the future.

Digital Multimeter

Even if you opt not to get a more complete toolbox, I strongly suggest getting a digital multimeter (DMM).

You can get a cheap one that covers the basic functionality for about what you'd pay for a good lunch. Of course, you can blow your whole week's food budget on an excellent DMM, but you shouldn't need to. As an embedded software engineer, you need only a few functions.

First, you need the voltage mode. Ideally, your DMM at least should be able to read from 0–20 V with 0.1 V granularity and from 0–2 V with 0.01 V granularity. Usually the question you are looking to answer with the voltage mode of the DMM is simple: are any volts getting there at all? A DMM with 1 mV granularity might be nice occasionally, but 80% of your DMM use will be to answer this broader question: "Is the chip or component even powered?"

The second most important mode is the resistance check mode, usually indicated with the Ohm symbol (Ω) and a series of three arcs. You probably don't need to know what value a resistor is; the real use for this mode is to determine when things are connected to each other. Just as voltage mode tests whether something has power, this mode will answer the question, "Are these things even talking to each other?"

To determine the resistance between two points on the board, the DMM sends a small current through the test points. As long as the board is off, this is safe. Don't run it in resistance mode with a powered board unless you know what you are doing.

When the DMM beeps, there is no significant resistance between the two test probes, indicating they are connected (you can check whether the beep is on by just touching the two probes together). Using this mode, your DMM can be used to quickly determine whether a cable or a trace has broken.

Finally, you want a DMM with a current mode denoted with an amp symbol (A or mA). This is for measuring how much power your system is taking. A good implementation of current mode can make a DMM a lot more expensive. Good news, though: if you need the fine granularity necessary for developing very low power products, you will need a tool that goes beyond the DMM. Got a DMM that can measure mA but realize the nA versions cost

more and probably aren't worth it. More on low power programming and the necessary tools are in Chapter 10.

Oscilloscopes and Logic Analyzers

Sometimes you need to know what the electrical signals on the board are doing. There are three flavors of scopes that can help you see the signals as they move:

Traditional oscilloscope

Measures analog signals, usually two or four of them at a time. A digital signal can be measured via analog, but it is kind of boring to see it in one of two spots. Of course, if your digital signal isn't high or low but somewhere in between, the oscilloscope will help you immensely.

Logic analyzer

Measures digital signals only, usually a lot of them at the same time (16, 32, or 64). At one time, logic analyzers were behemoth instruments that took days to set up, but generally found the problem within hours of setup being complete. Many modern logic analyzers hook to your computer and help you do the setup, so that this step takes only a little while. Additionally, many logic analyzers have *protocol analyzers* that interpret the digital bus so you can easily see what the processor is outputting. Thus, if you have an SPI communication bus and you are sending a series of bytes, a protocol analyzer will interpret the information on the bus. A *network analyzer* is a specific type of protocol analyzer, one that focuses on the complex traffic associated with a network. (However, an *RF network analyzer* is very different; it characterizes device response in the radio frequency range.)

Mixed signal oscilloscope

Combines the features of a traditional scope and logic analyzer with a couple of analog channels and 8 or 16 digital ones. A personal favorite of mine, the mixed signal scope can be used to look at different kinds of information simultaneously.

These scopes tend to be shared resources, as they are relatively expensive. You can buy reasonably priced ones that hook to your computer, but sometimes their feature sets are limited in non-obvious ways. Generally, you get what you pay for.

Setting Up a Scope

Step one is to determine which signal(s) are going to help you figure out a problem. Next, you'll need to attach the scope's ground clip to ground. If you have more than one (AC ground, DC ground, analog ground) and you aren't sure which one to connect the scope to, ask your electrical engineer. Choosing the wrong one can be detrimental to the health of your oscilloscope.

Next, attach the probes to the signals of interest. Many processors have such tiny pins that they require specialized probes. On the other hand, many hardware engineers put test points on their board, knowing that the software team is likely to need access to particular signals for debugging. Alternatively, you can get wires soldered on to the board at the signals you need and hook a probe to those.

Using a scope can be a bit daunting if you've never set one up before. Oscilloscope manuals vary, but they are the best place to look (and most of them are online). There is no generic manual to all of them, so I can only tell you the words to look for as you page through the manual.

Figure 3-13 shows a representation of an archetypal oscilloscope screen. The most important point is that time moves along the x-axis and voltage varies along the y-axis. The scales of these axes are configurable.

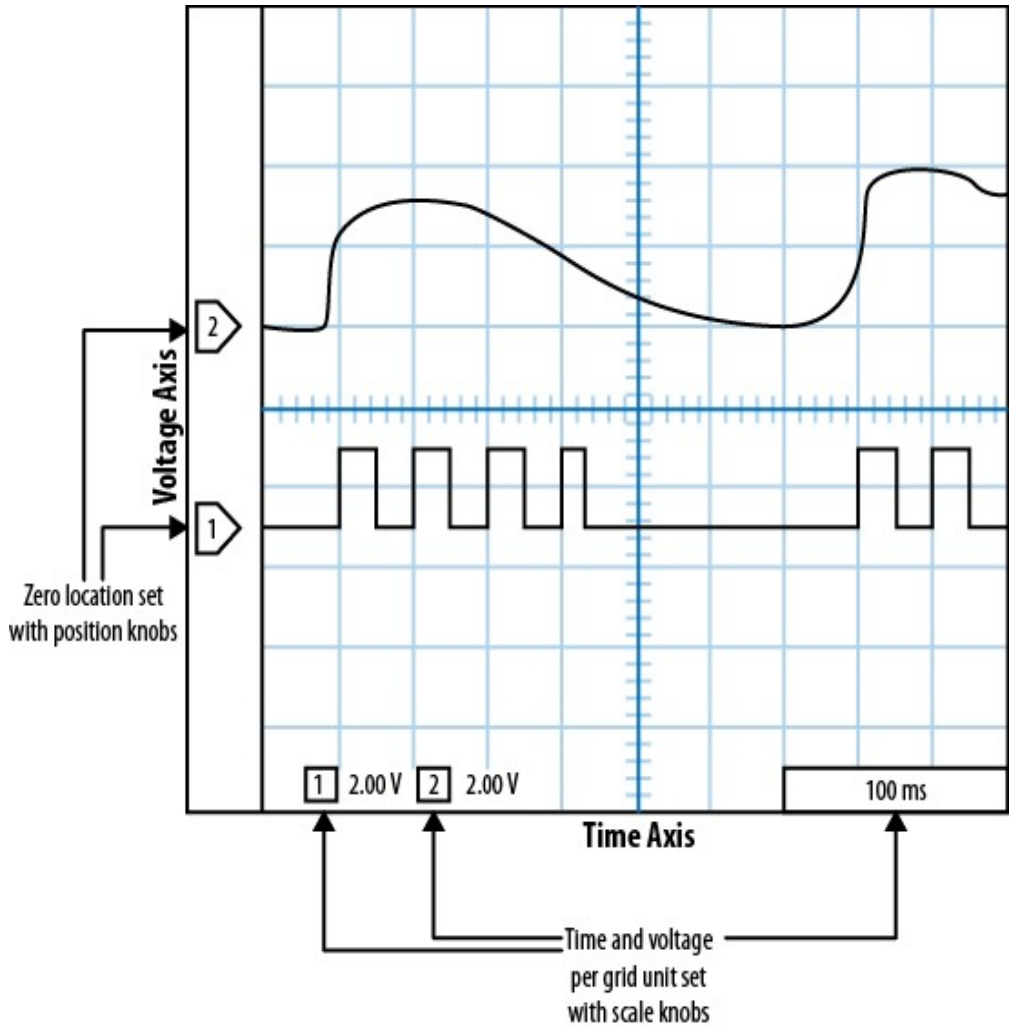


Figure 3-13. Simplified oscilloscope screenshot

Somewhere on the scope's interface, there should be a knob to make the timescale change. Move it to the right, and each horizontal tick goes from, say, 1 s to 0.5 s (and then down into the millisecond or microsecond range). This controls the timescale for the whole screen. For debugging, the goal is to start with the largest possible time base that shows what you are looking for.

WARNING

If you zoom in too far, you'll see strange things as the supposedly digital signals show their true (and weird) analog colors.

Once you set the time base, you'll need to set the scale of the voltage axis. This could be one knob that accesses all channels (or probes), making you

shift the channel some other way, or you might get one knob per channel. Either way, as you turn it to the right, each vertical block magnifies (5 V goes to 2 V, down to millivolts).

If you aren't sure, set the timescale to be about 100 ms/division and the voltage granularity to be about 2 V/block. You can zoom in (or out) from there if you need to.

A different knob will set where each channel is on the screen, in other words, where its zero line is. Figure 3-11 shows the zero line for each channel on the left side of the screen. There may be one knob per channel or a way to multiplex the knob between all the channels. Keep the channels a little offset from each other so you can see each one. (You can turn off the channels you don't need, probably with a button.)

Next, look for a knob to set the zero point of your timescale. This will cause a vertical line to wander your screen. Set it near the middle. Alternatively, you can set it toward the left of your screen if you are looking for information that happens after an event, or toward the right if you want to know what goes on before an event.

At this point, you can probably turn on your system and see a line wiggle. It may go by so quickly that you don't see what it does, but it can be a start. If nothing happens, look for the buttons that say Run/Stop. You want the scope to be in Run mode. The Stop mode freezes the display at a single point in time, which gives you the opportunity to think about what you've discovered. Near Run/Stop may be a button that says Single, which waits for a trigger and then puts the system in Stop mode. There may also be an Auto button, which will let the system trigger continuously.

WARNING

If there is an Auto button, be very careful about pushing it. Check to see whether the name means auto-trigger or auto-set. The former is useful, but the latter tries to configure your scope for you automatically. I find that this tends to reset the configuration to something completely random.

To set up a trigger, look for the trigger knob. This will put up a horizontal line to show where the trigger voltage level is. It is channel dependent, but unlike the other channel-dependent knob, usually you can't set a trigger for

each channel. Instead, it requires additional configuration, usually via an on-screen menu and button presses. You want the trigger to be on the channel that changes just at the start (or end) of an interesting event. You'll need to choose whether the trigger is activated going up or going down. You'll also need to choose how often the scope will trigger. If you want to see the first time it changes, set a long time-out. If you want to see the last time it changes, set a short one.

There are a few other things to note. Unless you know what you are doing, you don't want the scope in AC mode. And there is the possibility that the probes are giving signals that are 10x (or 1/10) the size on the screen. This depends on the probe, so if you are off by an order of magnitude, look for a 10x marker and toggle its state.

If you've set up your scope according to my instructions (and the scope manual) and it still doesn't work the way you want it to, get someone with more experience to help you. Scopes are powerful and useful tools, but using one well requires a lot of practice. Don't get discouraged if it is a bit frustrating to start.

Testing the Hardware (and Software)

I strongly recommend being ready to pull out the toolbox, DMM, and scope, but that can reasonably be left to your hardware engineer if you aren't ready to do it alone. As a software person, it is more important that you get as far as possible in building software that will test the hardware in a manner conducive to easy debugging.

Three kinds of tests are commonly seen for embedded systems. First, the *power-on self-test* (POST) runs every time you boot the system, even after the code is released. This test verifies that all of the hardware components are in working order and whatever else is needed to run your system safely. The more the POST tests, the longer the boot time is, so there is a trade-off that might impact the customer. Once the POST completes, the system is ready to be used by the customer.

NOTE

Ideally, all POST debug messages printed out at boot time should be accessible later. Whether it is the software version string or the type of sensor attached, there will be a time when you

require that information without power cycling.

The second sort of test should be run before every software release, but they might not be suitable to run at every boot, perhaps because they take too long to execute, return the system to factory default, or put unsightly test patterns on the screen. These tests verify the software and hardware are working together as expected.

These are my *unit tests*, though those words mean different things to different people. To me, it is automated test code that verifies a unit of source code is ready for use. Some developers want tests to cover all possible paths through the code. This can lead to unit test suites that are large and unwieldy, which is the argument used to avoid unit testing in embedded systems. My sort of unit tests are meant to test the basic functionality and the corner cases most likely to occur. This process lets me build testing into my development and keep it there, even after shipping.

WARNING

Find out what your industry (or your management) expects of your unit tests. If they mean for the tests to check all software paths, make sure you (and they) understand how much work and code that will take.

Some unit tests may be external to the hardware of the system (i.e., if you are using a sandbox to verify algorithms as suggested in Chapter 2). For the ones that aren't, I would encourage you to leave them in the production code if you can, making them accessible in the field upon some special set of criteria (for instance, "hold down these two buttons and cross your eyes as the system boots"). Not only will this let your quality department use the tests, but you may also find that using the unit tests as a first-line check in the field will tell you whether the system's hardware is acting oddly.

The third and final sorts of tests are those you create during bring-up, usually because a subsystem is not functioning as intended. These are sometimes throwaway checks, superseded by more inclusive tests or added to unit tests. Temporary bring-up code is OK. The goal of this whole exercise is not to write classic source code, but to build systems. And once you've checked

such code into your version control system, deleting it is fine because you can always recover the file if you realize you need the test again.

Building Tests

As noted earlier, the code to control peripherals (and the associated tests) often is written while the schematic is being completed. The good news is that you've just spent time with the datasheets, so you have a good idea how to implement the code. The bad news is that while you wait for the boards to arrive, you may end up writing drivers for six peripherals before you get to integrate your software with the hardware.

In Chapter 2, we had a system that communicated to a flash memory device via the SPI communication protocol (a partial schematic is reproduced in Figure 3-14). What do we need to test, and what tools do we need to verify the results?

- I/O lines are under software control (external verification with a DMM)
- SPI can send and receive bytes (external verification with a logic analyzer)
- Flash can be read and written (internal verification; use the debug subsystem to output results)

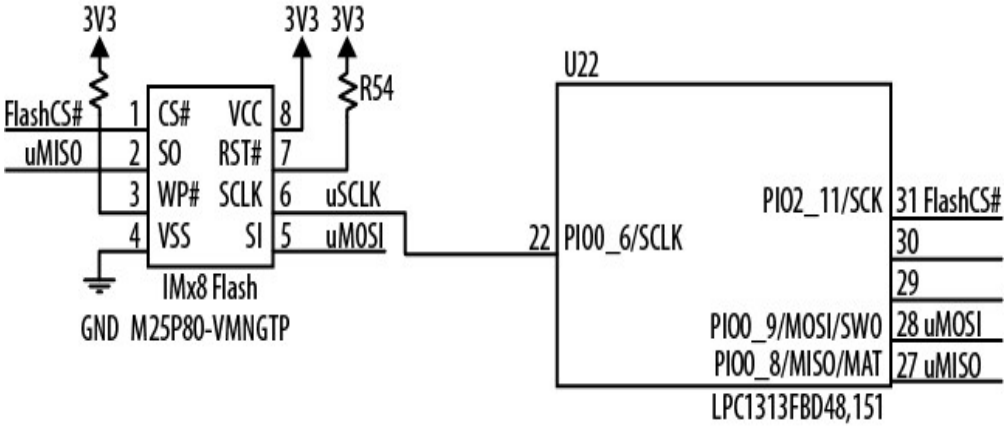


Figure 3-14. Flash memory schematic snippet

To make bring-up easy, you'll need to be able to do each of these. You might choose to run the all-inclusive flash test first. If that works, you know the

other two work. However, if it doesn’t, you’ll need the others available for debugging as necessary.

Other chapters cover controlling I/O lines (Chapter 4) and a bit more about SPI (Chapter 6), so for now let’s focus on the tests that we can write to test the flash memory. How flash works isn’t critical, but you can test your skills with datasheets by looking over the Macronix MX25V8035 flash memory datasheet (search for the part number on Google, Digikey, or the vendor’s website).

Flash Test Example

Flash memory is a type of nonvolatile memory (so it doesn’t get cleared when the power is turned off). Some other types of nonvolatile memory include ROM (read-only memory) and electrically erasable programmable read-only memory (EEPROM).

NOTE

Volatile memory doesn’t retain its value after a power cycle. There are different kinds of volatile memory, but they are all one type of RAM or another.

Like an EEPROM, flash can be erased and written to. However, most EEPROMs let you erase and write a byte at a time. With flash, you may be able to write a byte at a time, but you have to erase a whole sector in order to do so. The sector size depends on the flash (usually the larger the flash, the larger each sector). For our test’s component, a sector is 4 kilobytes (or 4096 bytes), and the whole chip contains 8 Mbit (1 Mbyte or 256 sectors). Flash usually has more space than an EEPROM and is less power hungry. However, EEPROMs come in smaller sizes, so they are still useful.

When we write a bring-up test, nothing in the flash chip needs to be retained. For the POST, we really shouldn’t modify the flash, as the system might be using it (for storing version and graphic data). For a unit test, we’d like to leave it the way we found it after testing, but we might be able to set some constraints on that.

Tests typically take three parameters: the goal flash address, so that the test can run in an unpopulated (or noncritical) sector; a pointer to memory; and

the memory length. The memory length is the amount of data the flash test will store to RAM. If the memory is the same size as the sector, no data will be lost in the flash. The following prototypes illustrate three types of test: an umbrella test that runs the others (and returns the number of errors it encounters), a test that tries to read from the flash (returning the number of bytes that were actually read), and a test that tries to write to the flash (returning the number of bytes written):

```
int FlashTest(uint32_t address, uint8_t *memory, uint16_t memLength);
uint16_t FlashRead(uint32_t addr, uint8_t *data, uint16_t dataLen);
uint16_t FlashWrite(uint32_t addr, uint8_t *data, uint16_t dataLen);
```

We'll use the `FlashTest` extensively during bring-up then put it in unit tests and turn it on as needed when things change or before releases. We don't want this test as part of the POST. Flash wears out after some number of write cycles (often 100,000; this information is in the datasheet). Further, at least two of these tests have the potential to be destructive to our data.

Plus, we don't need to do this testing on boot. If the processor can communicate with the flash at all, then it is reasonable to believe the flash is working. (You can check basic communication by putting a header in the flash that includes a known key, a version, and/or a checksum.)

There are two ways to access this flash part: individual bytes and multibyte blocks. Later, when running the real code, you'll probably want to use the faster block access. During initial bring-up and testing, you might want to start out with the simpler byte method to ensure a good foundation for your driver.

Test 1: Read existing data

Testing that you can read data from the flash actually verifies that the I/O lines are configured to work as a SPI port, that the SPI port is configured properly, and that you understand the basics of the flash command protocol.

For this test, we'll read out as much data as we can from the sector so we can write it back later. Here is the start of `FlashTest`:

```
  // Test 1: Read existing data in a block just to make sure it is possible
  dataLen = FlashRead(startAddress, memory, memLength);
  if (dataLen != memLength) {
    // read less than desired, note error
    Log(LogUnitTest, LogLevelError, "Flash test: truncation on byte read");
    memLength = dataLen;
```

```
  error++;
}
```

Note that there is no verification of the data, since we don't know if this function is being run with an empty flash chip. If you were writing a POST, you might do this read and then check that the data is valid (and then stop testing, because that is enough for a power-on check).

Test 2: Byte access

The next test starts by erasing the data in the sector. Then it fills the flash with data, writing one byte at a time. We want to write it with something that changes so we can make sure the command is effective. I like to write it with an offset based on the address. (The offset tells me that I'm not accidentally reading the address back.)

```
FlashEraseSector(startAddress);

// want to put in an incrementing value but don't want it to be the address
addValue = 0x55;
for (i=0; i< memLength; i++) {
  value = i + addValue;
  dataLen = FlashWrite(startAddress + i, &value, 1);
  if (dataLen != 1) {
    Log (LogUnitTest, LogLevelError, "Flash test: byte write error.");
    error++;
  }
}
```

To complete this check, you need to read the data back, byte by byte, and verify the value is as expected at each address (`address + addValue`).

Test 3: Block access

Now that the flash contains our newly written junk, confirm that block access works by putting back the original data. That means starting with an erase of the sector again:

```
FlashEraseSector(startAddress);
dataLen = FlashWrite(startAddress, memory, memLength);
if (dataLen != memLength) {
  LogWithNum(LogUnitTest, LogLevelError,
    "Flash test: block write error, len ", dataLen);
  error++;
}
```

Finally, verify this data using another byte-by-byte read. We know that works from test 2. If the results are good, then we know that block writes work from this test and that block reads work from test 1. The number of errors is returned to the higher-level verification code.

NOTE

This tests the flash driver software. It doesn’t check that the flash has no sticky bits (bits that never change, even though they should). A manufacturing test can confirm that all bits change, but most flash gets verified that way before it gets to you.

Test wrap-up

If the board passes these three tests, you can be confident that the flash hardware and software both work, satisfying your need for a bring-up test and a unit test.

For most forms of memory, the pattern we’ve seen here is a good start:

Read the original data.

Write some changing but formulaic junk.

Verify the junk.

Rewrite the original.

Verify the original.

However, there are many other types of peripherals, more than I can cover here. Although automated tests are best, some will need external verification, such as an LCD that gets a pattern of colors and lines to verify its driver. Some tests will need fake external inputs so that a sensing element can be checked.

For each of your peripherals and software subsystems, try to figure out what test will give you the confidence that it is working reliably and as expected. It doesn’t sound difficult, but it can be. Designing good tests is one of those things that can make your software great.

Command and Response

Let’s say you take the advice in the previous section and create test functions for each piece of your hardware. During bring-up, you’ve made a special image that executes each test on power-up. If one of them fails, you and the hardware engineer may want to run just that test over and over. So you recompile and reload. Then you want to do the same thing for a different test. And yet another one. Wouldn’t it be easier to send your embedded system commands and have it run the tests as needed?

Embedded systems often do not have the rich user interface experience found in computers (or even smartphones). Many are controlled through a command line. Even those with screens often use a command-line interface for debugging. The first part of this section describes how to send commands in C using function pointers in a command-handling jump table. This nifty problem-and-solution gives me an opportunity to show the standard command pattern, which is a useful pattern to know, whatever language you are using.

Figure 3-15 shows some high-level goals for our automated command handler, one that will let us send a command from a serial terminal on the PC and get a response from the unit under test.

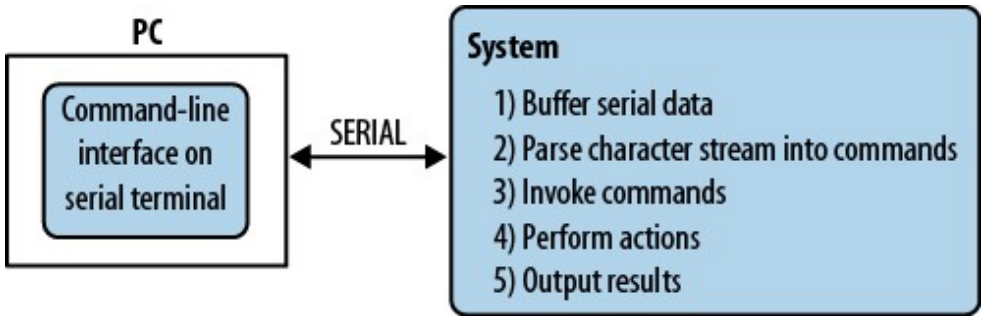


Figure 3-15. Goals for a command handler

Once you have some data read in, the code will need to figure out which function to call. A small interpreter with a command table will make the lives of those around you easier. By separating the interface from the actual test code, you will be able to extend it more easily in unforeseen directions.

Creating a command

Let's start with a small list of commands you want to implement:

Version of the code

: Outputs the version information

Test the flash

: Runs the flash unit test, printing out the number of errors upon completion

Blink LED

: Sets an LED to blink at a given frequency

Help

Lists available commands with one-line descriptions

Once you have a few commands implemented, adding commands as you go along should be pretty easy.

I'm going to show how to do this in C because it is probably the scariest way you'd implement it. Object-oriented languages such as C++ and Java offer friendlier ways to implement this functionality using objects. However, the C method will be smaller and generally faster, so factor that in when you are choosing how to implement this pattern. For readers unfamiliar with the C language's function pointers, the sidebar "Function Pointers Aren't So Scary" provides an introduction.

Each command we can call will be made up of a name, a function to call, and a help string:

```
typedef void(*functionPointerType)(void);
struct commandStruct {
  char const *name;
  functionPointerType execute;
  char const *help;
};
```

An array of these will provide our list of commands:

```
const struct commandStruct commands[] ={
  {"ver", CmdVersion,
    "Display firmware version"},
  {"flashTest", CmdFlashTest,
    "Runs the flash unit test, prints number of errors upon completion"},
  {"blinkLed", CmdBlinkLed,
    "Sets the LED to blink at a desired rate (parameter: frequency (Hz))"},
  {"help", CmdHelp,
    "Prints out help messages"},
  {"",0,""} //End of table indicator. MUST BE LAST!!!
};
```

The command execution functions `CmdVersion`, `CmdFlashTest`, `CmdBlink`, and `CmdHelp` are implemented elsewhere. The commands that take parameters will have to work with the parser to get their parameters from the character stream. Not only does this simplify this piece of code, but it also gives greater flexibility, allowing each command to set the terms of its use, such as the number and type of parameters.

Note that the list in the `help` command is a special command that prints out the name and help strings of all items in the table.

FUNCTION POINTERS AREN'T SO SCARY

Can you imagine a situation where you don't know what function you want to run until your program is already running? For example, say you wanted to run one of several signal-processing algorithms on some data coming from your sensors. You can start off with a `switch` statement controlled by a variable:

```
switch (algorithm) {
case kFIRFilter:
return fir(data, dataLen);
case kIIRFilter:
return iir(data, dataLen);
...
}
```

Now when you want to change your algorithm, you send a command or push a button to make the algorithm variable change. If the data gets processed continually, the system still has to run the `switch` statement, even if you didn't change the algorithm.

In object-oriented languages, you can use a reference to an interface. Each algorithm object would implement a function of the same name (for this signal-processing example, we'd call it `filter`). Then when the algorithm needed to change, the caller object would change. Depending on the language, the interface implementation could be indicated with a keyword or through inheritance.

C doesn't have those features (and using inheritance in C++ may be verboten in your system due to compiler constraints). So we come to function pointers, which can do the same thing.

To declare a function pointer, you need the prototype of a function that will go in it. For our `switch` statement, that would be a function that took in a data pointer and data length and didn't return anything. However, it could return results by modifying its first argument. The prototype for one of the functions would look like this:

```
void fir(uint16_t* data, uint16_t dataLen);
```

Now take out the name of the function and replace it with a star and a generic name surrounded by parentheses:

```
void (*filter)(uint16_t* data, uint16_t dataLen);
```

Instead of changing your algorithm variable and calling the `switch` statement to select a function, you can change the algorithm only when needed and call the function pointer. There are two ways to do this, the implicit method (on the left below) and the explicit method (right). They are equivalent, only two different forms of syntax.

```
filter = fir; // or filter = &fir;
filter(data, dataLen); // or (*filter)(data,dataLen);
```

Once you are comfortable with the idea of function pointers, they can be a powerful asset when the code needs to adapt to its environment. Some common uses for function pointers include the command structure described in this chapter, callbacks to indicate a completed event, and mapping button presses to context-sensitive actions.

One caution: excessive use of function pointers can cause your processor to be slow. Some processors try to predict where your code is going and load the appropriate instructions before execution. Function pointers inhibit branch prediction because the processor can’t guess where you’ll be after the call.

However, other methods of selecting on-the-fly function calls also interrupt branch prediction (such as the `switch` statement we started with). Unless you have reached the stage of hand-tuning the assembly code, it generally isn’t worth worrying about the slight slowdown caused by the awesome powers of the function pointer.

Invoking a command

Once the client on the command line indicates which command to run by sending a string, you need to choose the command and run it. To do that, go through the table, looking for a string that matches. Once you find it, call the function pointer to execute that function.

Compared to the previous section, this part seems almost too easy. That is the goal. Embedded systems are complex, sometimes hideously so, given their tight constraints and hidden dependencies. The goal of this (and many other patterns) is to isolate the complexity. You can’t get rid of the complexity; a system that does nothing isn’t very complex, but it isn’t very useful either.

There is still a fair amount of complexity in setting up the table and the parsing code to use it, but those details can be confined to their own spaces.

Command Pattern

What we've been looking at is a formal, classic design pattern. Where the command handler I've described is a tactical solution to a problem, the *command pattern* is the strategy that guides it and other designs like it.

The overarching goal of this pattern is to decouple the command processing from the actual action to be taken. The command pattern shows an interface for executing operations. Any time you see a situation where one piece of a system needs to make requests of another piece without the intermediaries knowing the contents of those requests, consider the command pattern.

As shown in Figure 3-16, there are four pieces:

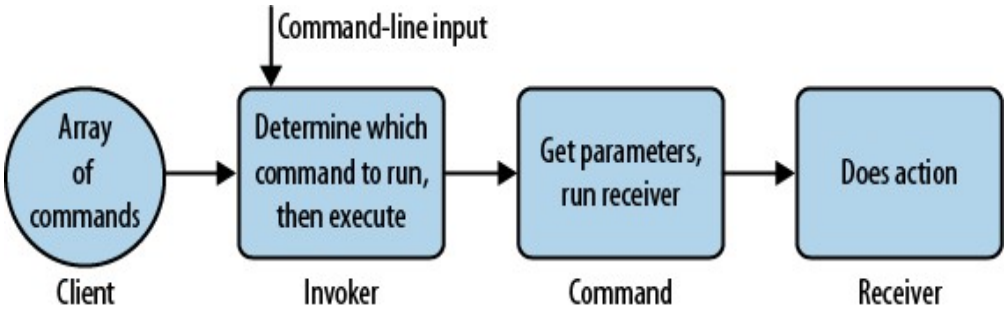


Figure 3-16. How the command handler works

Client

A table that maps the receivers onto commands. Clients create concrete command objects and create the association between receivers and command objects. A client may run at initialization (as was done in our example by creating an array) or create the associations on the fly.

Invoker

Parser code that determines when the command needs to run and executes it. It doesn't know anything about the receiver. It treats every command the same.

Command

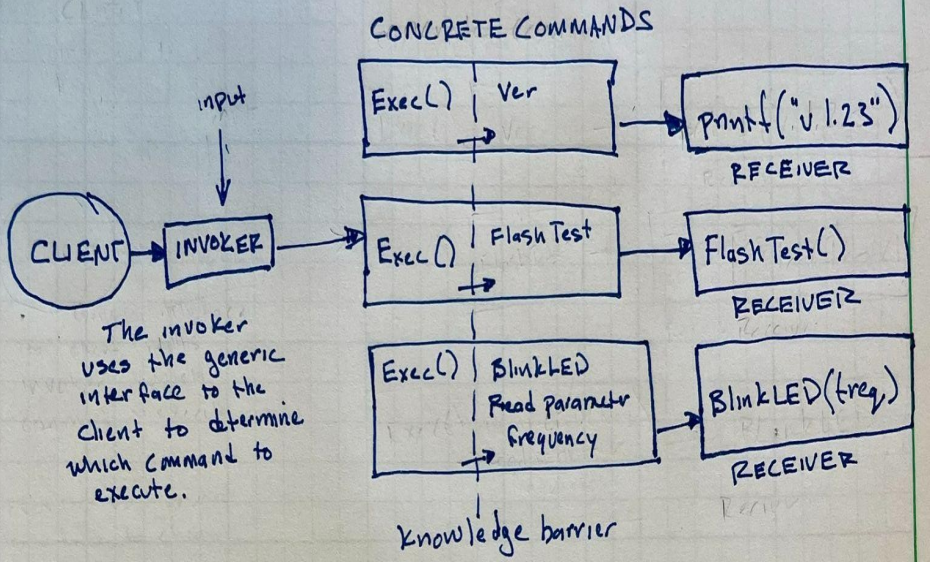
An item from the client table that is an interface for executing an operation. This is a C++ class, a Java interface, or a C structure

with a function pointer. An instantiation of the command interface is called a *concrete command*.

Receiver

A function that knows how to service the request. Running the receiver code is the goal of the person (or other device) that sent the command.

Figure 3-17 shows the purpose of each element of the command pattern.



Modified Fig 3-15: Command pattern and the knowledge barrier

NOTE: changed diagram's horizontal direction to match Z-14's

DESCRIPTION

The CLIENT points to the INVOKER with a note below saying: "The INVOKER uses the generic interface to the CLIENT to determine which command to execute."

Input also points to the INVOKER. The INVOKER points to the center option of three CONCRETE COMMANDS. Each CONCRETE COMMAND has Exec() on the invoker side and a function call on the other. These are separated by a dashed line indicating the knowledge barrier.

The first CONCRETE COMMAND is Ver which points to a RECEIVER that calls printf("v 1.23"). The second COMMAND says FlashTest which points to a Flash Test RECEIVER.

The last COMMAND is BlinkLED which has a note that it needs the parameter frequency. The COMMAND then points to a receiver that says BLINKLED(freq).

Figure 3-17. Command pattern and the knowledge barrier

NOTE

Consider the client as protecting the secrets of the system by disguising all of the commands to look alike. This decouples the commands from your code making it much easier to expand later (which you will need to do).

Dealing with Errors

The longevity of code shocks me. For all that it sometimes seems like we are rewriting the same old things in different ways, one day you may discover that a piece of code you wrote at a start-up over a decade ago is being used by a Fortune 500 company. Once it works well enough, why fix the depths of code?

It only makes it scarier to know that at some point your code will fail. An error will occur, either from something you wrote or from an unexpected condition in the environment. There are two ways to handle errors. First, the system can enter a state of *graceful degradation*, where the software does the best it can. Alternatively, the system could fail loudly and immediately. Long-term sensor-type systems require the former, whereas medical systems require the latter. Either way, a system should fail safely.

But how to implement either one? More importantly, what criteria should be used to determine which subsystems implement which error-handling method? What you do depends on your product; my goal is to get you to think about error handling during design.

Consistent Methodology

Functions should handle errors as best they can. For example, if a variable can be out of range, the range should be fixed and the error logged as appropriate. Functions can return errors to allow a caller to deal with the problems. The caller should check and deal with the error, which may mean passing it further upstream in a multilayered application. In many cases, if the error is not important enough to check, it is not important enough to return. On the other hand, there are cases where you want to return a diagnostic code that is used only in testing. It can be noted in the comments

that the returned error code is not for normal usage or should be used only in `assert()` calls.

`assert()` is not always implemented in embedded systems, but it is straightforward to implement in a manner appropriate to the system. This might be a message printed out on the debugger console, a print to a system console or log, a breakpoint instruction such as `BKPT`, or even an I/O line or LED that toggles on an error condition. Printing changes the embedded system's timing, so it is often beneficial to separate the functions for error communications to allow other methods of output (e.g., an LED).

NOTE

A breakpoint instruction tells the processor to stop running if the debugger is attached. A *programmatic breakpoint* can be compiled into your code. Usually it is an assembly function, often hidden by a macro:

```
#define BKPT() __asm__("BKPT")
#define BKPT() __asm__("bkpt #0")
#define BKPT() __asm__("break")
```

NOTE

These breakpoints get compiled into your code (they don't count against your limited hardware breakpoint budget!). This can make it easier to walk through code, particularly if you are seeing errors only upon certain conditions.

Error return codes for an application or system should be standardized over the code base. A single high-level *errorCodes.h* file (or some such) can be created to provide consistent errors in an enumerated format. Some suggested error codes are:

- No error (should always be 0)
- Unknown error (or unrecognized error)
- Bad parameter
- Bad index (pointer outside range or null)

- Uninitialized variable or subsystem
- Catastrophic failure (this may cause a processor reset unless it is in a development mode, in which case it probably causes a breakpoint or spin loop)

There should be a minimum number of errors—generic errors—so that the application can interpret them. Although specificity is lost (`UART_FAILED_TO_INIT_BECAUSE_SECOND_PARAMETER_WAS_TOO_HIGH`), the generalization makes error handling and usage easier (if the error `PARAMETER_BAD` occurs in a subsystem, you have a good place to start to look for it). Essentially, by keeping it simple, you make sure to hand the developer the important information (the existence of a bug), and additional debugging can dig into where and why.

Error Checking Flow

Returning errors doesn't do anyone any good if the errors aren't handled. Too often we code for the working case, ignoring the other paths. Are we hoping that the errors just never occur? That's silly.

On the other hand, it gets pretty tedious to handle individual errors when the function should skip to the end and return its own error, bubbling up the problem to a higher level. While some folks use a nested if/else statement, I usually prefer to flow through the function, trying to keep the error redundant enough that it is generally ignorable in the good path flow.

```
error = FunctionSay();
if (error == NO_ERROR) {
  error = FunctionHello();
}
if (error == NO_ERROR) {
  error = FunctionWorld();
}
if (error != NO_ERROR) {
  // handle error state
}
return error;
```

This allows you to call several functions at a time, checking the error at the end instead of at each call.

Error-Handling Library

An error-handling library is another option.

```
ErrorSet(&globalErrorCode, error);
```

or

```
ErrorSet(&globalErrorCode, FunctionFoo());
```

The `ErrorSet` function would not overwrite a previous error condition if this function returned without a failure. As before, this allows you to call several functions at a time, checking the error at the end instead of at each call.

In such an error-handling library, there would be four functions, implemented and used as makes sense with the application: `ErrorSet`, `ErrorGet`, `ErrorPrint`, and `ErrorClear`. The library should be designed for debugging and testing, though the mechanism should be left in place, even when development is complete. For example, `ErrorPrint` may change from writing out a log of information on a serial port to a small function that just toggles an I/O line. This is not an error for the final user to deal with; it is for a developer confronted with production units that do not perform properly.

NOTE

If this sounds vaguely familiar, it may be because this is what standard C/C++ libraries do. If your file fails to open with `fopen`, not only does it return NULL to tell you that an error occurred, `fopen` also puts an error code into `errno` with more information about what went wrong. Even for most embedded systems, you can use `perror` and `strerror` to get text versions of the error code. Of course, as with my version above, if you don't clear `errno`, you may be looking at old errors.

Debugging Timing Errors

When debugging hardware/software interactions (or any software that is time critical), serial output such as logging or `printf` can change the timing of the code. In many cases, a timing problem will appear (or disappear) depending on where your output statements and/or breakpoints are located. Debugging becomes difficult when your tools interfere with the problem.

One of the advantages to using an error library (or a logging library, as described in Chapter 2) is that you don't have to output the data; you can store it locally in RAM, which is usually a much faster proposition. In fact, if you are having trouble with timing, consider making a small buffer (4–16

bytes, depending on your RAM availability) to contain signals from the software (one or two bytes). In your code, fill the buffer at interesting trigger points (e.g., an `ASSERT` or `ErrorSet`). You can dump this buffer when the code exits the time-sensitive area. If you are most interested in the last error, use a circular buffer to continually capture the last few events (circular buffers are discussed in more detail in Chapter 6).

Alternatively, if you have some input into the board design, I highly recommend pushing for spare processor I/Os to be available on the board and easily accessible with a header. They will come in useful for debugging (especially tricky timing issues where serial output breaks the timing), showing the status of the system in testing, and profiling processor cycles. These are called *test points*. The best EEs will ask you how many you need as they work on the schematic. There will be more on their usage in Chapter 8.

Further Reading

If you want to learn more about handling hardware safely, reading schematics, and soldering, I suggest getting the latest edition of *Make: Electronics* by Charles Platt, published by O'Reilly. This step-by-step introduction is an easy read, though better if you follow along with components in hand.

For more information about getting from prototype to shipping units, I recommend Alan Cohen's *Prototype to Product: A Practical Guide for Getting to Market*. It goes through all of the steps needed to go from a napkin sketch to a product.

Along those lines but more focused on your personal career is Camille Fournier's *The Manager's Path: A Guide for Tech Leaders Navigating Growth and Change*. Even if you aren't a manager, this book talks about the things managers need to know which will give you insight into how to present your work. This is the sort of book I read for a few chapters then put on my shelf until I receive a promotion, then read the next few chapters.

Adafruit and Sparkfun are amazing resources, filled with well-documented tutorials combining hardware and software. Even better, their extensive github repositories have **good** code. If they have code for a part similar to the one you are using, their code may hold the key to getting your driver to work.

INTERVIEW QUESTION: TALKING ABOUT FAILURE

Tell me about a project that you worked on that was successful. Then, tell me about a project that you worked on that was not so successful. What happened? How did you work through it? (Thanks to Kristin Anderson for this question.)

The goal of this question is not really about judging an applicant's previous successes and failures. The question starts there because the processes that lead to success (or failure) depend on knowledge, information, and communication. By talking about success or failure, the applicant reveals what he learns from available information, how he seeks out information, and how he communicates important information to others on the team. If he can understand the big picture and analyze the project's progress, he's more likely to be an asset to the team.

The successful half of the question is interesting to listen to, particularly when the interviewee is excited and passionate about his project. I want to hear about his part in making it successful and the process he used. However, like many of the more technical questions, the first half of the question is really about getting to the second half of the question: failure.

Did he understand all the requirements before jumping into the problem, or were there perhaps communication gaps between the requirements set by the marketing group and the plans developed by the engineers? Did the applicant consider all the tasks and even perhaps Murphy's Law before developing a schedule and then add some time for risk?

Did he bring up anything relative to any risks or issues he faced and any mitigation strategies he considered? Some of these terms are program management terms, and I don't really expect the engineers to use all the right buzzwords. But I do expect them to go beyond talking about the technical aspects and be able to tell me other factors in successful projects.

Often, in talking about successful projects, an applicant will say he understood the requirements and knew how to develop the right project and make people happy. That is enough to go on to the next question.

Then, when I ask what went wrong, I really want to hear how the applicant evaluated the factors involved in trying for a successful project

and whether he can tell me why he thought the project failed (poor communication, lack of clear goals, lack of clear requirements, no management support, etc.). There is no right or wrong answer; what I'm interested in is how he analyzes the project to tell me what went wrong. (Though if he only takes the time to blame everyone around him, I do get worried.) I am disappointed by the interviewees who cannot think of an unsuccessful project, because no project goes perfectly, so I tend to think they weren't paying attention to the project as a whole. Understanding more than just what technical work he has to do makes an engineer a much better engineer.

Chapter 4. Outputs, Inputs, and Timers

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at s*evans@oreilly.com*.

Inputs, outputs, and timers form the basis for almost everything embedded systems do. Even the communication methods to talk to other components are made up of these (see bit-banging drivers in the next chapter). In this chapter, I'm going to walk through a product development example. The goals will constantly change so it will feel like real life. That also lets us explore how to go from a simple system of a blinking light to debouncing buttons and dimming LEDs. Along the way, we'll see a lot about timers and how they do much more than measure time.

However, before we get started with world domination, err, I mean product development, you need to know a bit about registers, the interface from your software to the processor.

Handling Registers

To do anything with an I/O line, we need to talk to the appropriate register. As described in "Reading a Datasheet", you can think of registers as an API to the hardware. Described in the chip's user manual, registers come in all flavors to configure the processor and control peripherals. They are memory-mapped, so you can write to a specific address to modify a particular register. Often each bit in the register means something specific which means we need to start using binary numbers to look at the values of specific bits.

Binary and Hexadecimal Math

Becoming familiar with basic binary and hexadecimal (hex) math will make your career in embedded systems far more enjoyable. Shifting individual bits around is great when you need to modify only one or two places. But if you need to modify the whole variable, hex comes in handy because each digit in hex corresponds to a nibble (four bits) in binary. (Yes, of course a nibble is half of a byte. Embedded folks like their puns.)

Binary	Hex	Decimal	Remember this number
0000	0	0	This one is easy.
0001	1	1	This is (1 << 0).
0010	2	2	This is (1<< 1). Shifting is the same as multiplying by $2^{shiftValue}$.
0011	3	3	Notice how in binary this is just the sum of one and two.
0100	4	4	(1 << 2) is a 1 shifted to the left by two zeros,
0101	5	5	This is an interesting number because every other bit is set.
0110	6	6	See how this looks like you could shift the three over to the left by one? This could be put together as ((1<<2)|(1<<1)), or ((1<<2) + (1<<1)), or, most commonly, (3 << 1)).
0111	7	7	Look at the pattern of binary bits. They are very repetitive. Learn the pattern, and you'll be able to generate this table if you need to.
1000	8	8	(1 << 3). See how the shift and the number of zeros are related? If not, look at the binary representation of 2 and 4.
1001	9	9	We are about to go beyond the normal decimal numbers. Because there are more digits in hexadecimal, we'll borrow some from the alphabet. In the meantime, 9 is just 8 + 1.
1010	A	10	This is another special number with every other bit set.

1011	B	11	See how the last bit goes back and forth from 0 to 1? It signifies even and odd.
1100	C	12	Note how C is just 8 and 4 combined in binary? So of course it equals 12.
1101	D	13	The second bit from the right goes back and forth from 0 to 1 at half the speed of the first bit: 0, then 0, then 1, then 1, then repeat.
1110	E	14	The third bit also goes back and forth, but at half the rate of the second bit.
1111	F	15	All of the bits are set. This is an important one to remember.

Note that with four bits (one hex digit) you can represent 16 numbers, but you can't represent the number 16. Many things in embedded systems are zero-based, including addresses, so they map well to binary or hexadecimal numbers.

A byte is two nibbles, the left one being shifted up four spaces from the other. So 0x80 is (0x8 << 4).

A 16-bit word is made up of two bytes. We can get that by shifting the most significant byte up 8 bits, then adding it to the lower byte.

```
0x1234 = (0x12 << 8) + (0x34)
```

A 32-bit word is 8 characters long in hex but 10 characters long in decimal. While this dense representation can make debug prints useful, the real reason we use hex is to make binary easier.

TIP

Since memory is generally viewed in hex, some values are used to identify anomalies in memory. In particular, expect to see (and use) 0xDEADBEEF or 0xDEADC0DE as an indicator (they are a lot easier to remember in hex than in decimal). Two other important bytes are 0xAA and 0x55. Because the bits in these numbers alternate, they are easy to see on an oscilloscope and good for testing when you want to see a lot of change in your values.

Bitwise Operations

As you work with registers, you will need to think about things at the bit level. Often you'll turn specific bits on and off. If you've never used bitwise operations, now is the time to learn. Where `&&` commonly means AND, `!` means NOT, and `||` means OR, these work on the idea that anything zero is false and anything non-zero is true.

Bitwise operations operate on a bit by bit basis. For AND, if the two inputs have a bit set, then the output will as well. Other bits in the output will be zero. For OR, if either of the two inputs has a bit set, then the output will as well.

Bitwise Operation	Meaning	Syntax	Examples

AND	If both of the two inputs have a bit set, the output will as well	&	`0x01 & 0x02 = 0x00`
			`0x01 & 0x03 = 0x01`
			`0xFF & 0x00 = 0x00`

OR	If either of the two inputs has a bit set, the output will as well	|	`0x01 | 0x02 = 0x03`
			`0x01 | 0x03 = 0x03`
			`0xFF | 0x00 = 0xFF`

XOR	If only one of the two inputs has a bit set, the output will as well	^	`0x01 ^ 0x02 = 0x03`
			`0x01 ^ 0x03 = 0x02`
			`0xAA ^ 0xF5 = 0x5F`

NOT	Every bit is set to its opposite	~	`~0x01 = 0xFE ~0x00 = 0xFF ~0x55 = 0xAA`

XOR

XOR (exclusive or) is a somewhat magical bitwise operation. It doesn't have a logical analog and isn't used as much as the others we've seen, so remembering it can be tough.

Imagine your nemesis puts on his blog that he is going to the movies this evening, which is just what you had planned to do. If you both avoid going, the movie won't play, because there won't be an audience. One of you can go to see the movie. But if you both go, the movie won't play (the theater is still not happy about the fuss you caused last time).

The truth table looks like:

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

NOTE

XOR is often represented as a Venn diagram, as shown in Figure 4-1. Note that XOR is true inside each circle, but not where they overlap.

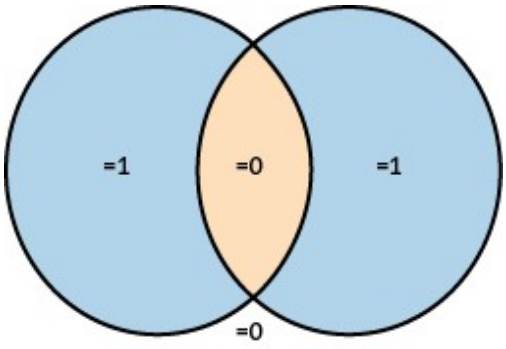


Figure 4-1. XOR Venn diagram

XOR has some nifty applications in computer graphics and finding overflows (math errors).

If it is Input A, we are using only the bottom half of the chart. If the register (Input B) already has that pin set, what would the output be?

Test, Set, Clear, and Toggle

Hex and bitwise operations are only a lead up into being able to interact with registers. If you want to know if a bit is set, you need to bitwise AND with the register:

```
test = register & bit;
test = register & (1 << 3); // check 3rd bit in the register
test = register & 0x08;    // same, different syntax
```

Note that the `test` variable will either equal zero (bit not set in the register) or 0x04 (bit is set). It isn't true and false, though you can use it in normal conditionals as they only check for zero or non-zero.

If you want to set a bit in a register, OR it with the register:

```
register = register | bit;
register = register | (1 << 3); // turn on the 3rd bit in the register
register |= 0x08;                 // same, different syntax
```

Clearing a bit is quite a bit more confusing because you want to leave the other bits in the register unchanged. The typical strategy is to invert the bit we want to clear using bitwise NOT (~). Then AND the inverted value with the register. This way, the other bits remain unchanged.

```
register = register & ~bit;
register = register & ~(1 << 3); // turn off the 3rd bit in the register
register &= ~0x08;              // same, different syntax
```

If you wanted to toggle a bit you could check its value and then set or clear it as needed. For example:

```
test = register & bit;
if (test) {                         // bit is set, need to clear it
    register = register & ~bit;
} else {                           // bit is not set, need to set it
    register = register | bit;
}
```

However, there is another way to toggle a bit using the xor operation:

```
register = register ^ bit;
register = register ^ (1 << 3);
```

See, if the register has the bit set, this will have two ones and will then be set to zero. The other bits in the register won't be changed because only the single bit is set in the second input, the others are zero.

Enough review. You will need to know these operations to use registers. If this isn't something you are comfortable with, there are resources at the end of the chapter for you to learn more or practice.

Toggling an Output

Marketing has come to you with an idea for a product. When you see through the smoke and mirrors, you realize that all they need is a light to blink.

Most processors have pins whose digital states can be read (input) or set (output) by the software. These go by the name of I/O pins, general-purpose I/O (GPIO), digital I/O (DIO), and occasionally general I/O (GIO). The basic use case is usually straightforward, at least when an LED is attached:

1. Initialize the pin to be an output (as an I/O pin, it could be input or output).

2. Set the pin high when you want the LED on. Set the pin low when you want the LED off. (Although the LED also can be connected to be on when the pin is low, but this example will avoid inverted logic.)

Through this chapter, I'll give you examples from three different user manuals so you get an idea of what to expect in your processor's documentation. Microchip's ATtiny AVR microcontroller manual describes an 8-bit microcontroller with plenty of peripherals. The TI MSP430x2xx User's Guide describes a 16-bit RISC processor designed to be ultra low power. The STMicroelectronics STM32F103xx Reference Manual describes a 32-bit ARM Cortex microcontroller. You won't need these documents to follow along, but I thought you might like to know the processors the examples are based upon.

Set the Pin to Be an Output

Getting back to marketing's request to toggle an LED, most I/O pins can be either inputs or outputs. The first register you'll need to set will control the direction of the pin so it is an output. First, determine which pin you will be changing. To modify the pin, you'll need to know the pin name ("I/O pin 2," not its number on the processor (for example, pin 12). The names are often inside the processor in schematics where the pin number is on the outside of the box (as shown in Figure 4-2).

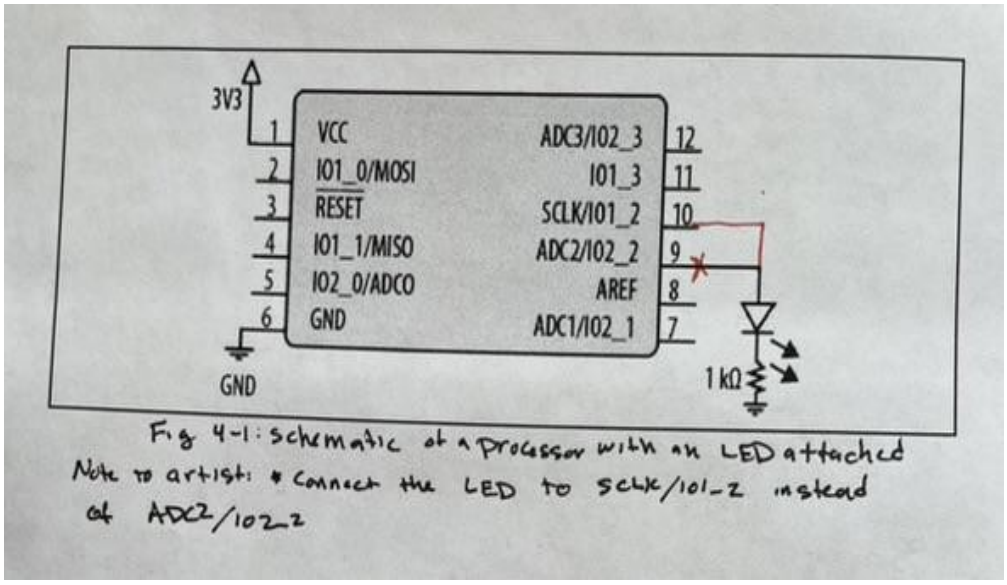


Figure 4-2. Schematic of a processor with an LED attached

The pins may have multiple numbers in their name, indicating a port (or bank) and a pin in that port. (The ports may also be letters instead of numbers.) In the figure, the LED is attached to processor pin 10, which says "SCLK/IO1_2." This pin is shared between the SPI port (remember, this is a communication method, discussed in Chapter 6) and the I/O subsystem (IO1_2). The user manual will tell you whether the pin is an I/O by default or a SPI pin (and how to switch between them). Another register may be needed to indicate the purpose of the pin. Most vendors are good about cross-referencing the pin setup, but if it is shared between peripherals, you may need to look in the peripheral section to turn off unwanted functionality. In our example, we'll say the pin is an I/O by default.

In the I/O subsystem, it is the second pin (2) of the first bank (1). We'll need to remember that and to make sure the pin is used as an I/O pin and not as a SPI pin. Your processor user manual will describe more. Look for a section with a name like "I/O Configuration," "Digital I/O Introduction," or "I/O Ports." If you have trouble finding the name, look for the word "direction," which tends to be used to describe whether you want the pin to be an input or an output.

Once you find the register in the manual, you can determine whether you need to set or clear a bit in the direction register. In most cases, you need to set the bit to make the pin an output. You could determine the address and hardcode the result:

```
*((int*)0x0070C1) |= (1 << 2);
```

However, please don't do that.

The processor or compiler vendor will almost always provide a header or hardware abstraction layer (HAL) that hides the memory map of the chip so you can treat registers as global variables. If they didn't give you a header, make one for yourself so that your code looks more like one of these lines:

STM32F103 processor

```
GPIOA->CRL |= 1 << 8;  // set IOA_2 to be an output
```

MSP430 processor

```
P1DIR |= BIT2;         // set IO1_2 to be an output
```

ATtiny processor

```
DDRB |= 0x4;           // set IOB_2 to be an output
```

Note that the register names are different for each processor, but the effect of the code in each line is the same. Each processor has different options for setting the second bit in the byte (or word).

In each of these examples, the code is reading the current register value, modifying it, and then writing the result back to the register. This read-modify-write cycle needs to happen in atomic chunks. If you read the value, modify it, and then do some other stuff before writing the register, you run the risk that the register has changed and the value you are writing is out of date. The register modification will change the intended bit, but also might have unintended consequences.

Turn On the LED

The next step is to turn on the LED. Again, we'll need to find the appropriate register in the user manual.

STM32F103 processor

```
GPIOA->ODR |= (1 << 2);    // IOA_2 high
```

MSP430 processor

```
P1OUT |= BIT2;              // IO1_2 high
```

ATtiny processor

```
PORTB |= 0x4;               // IOB_2 high
```

The GPIO hardware abstraction layer header file provided by the processor vendor shows how the raw addresses get masked by some programming niceties. In STM32F103x6, the I/O registers are accessed at an address through a structure (I've made some reorganization and simplifications to the file):

```
typedef struct
{
  __IO uint32_t CRL;    // Port configuration (low)
  __IO uint32_t CRH;    // Port configuration (high)
  __IO uint32_t IDR;    // Input data register
  __IO uint32_t ODR;    // Output data register
  __IO uint32_t BSRR;    // Bit set/reset register
  __IO uint32_t BRR;    // Bit reset register
  __IO uint32_t LCKR; // Port configuration lock
} GPIO_TypeDef;

#define PERIPH_BASE       0x40000000UL
#define APB2PERIPH_BASE   (PERIPH_BASE + 0x00010000UL)
```

```
#define GPIOA_BASE        (APB2PERIPH_BASE + 0x00000800UL)
#define GPIOA             ((GPIO_TypeDef *)GPIOA_BASE)
```

The header file describes many registers we haven't looked at. These are also in the user manual section, with a lot more explanation. I prefer accessing the registers via the structure because it groups related functions together, often letting you work with the ports interchangeably.

Once we have the LED on, we'll need to turn it off again. You just need to clear those same bits, as shown in the register section ("Test, Set, Clear, and Toggle"):

STM32F103 processor

```
          GPIOA->ODR &= ~(1 << 2);          // IO1_2 low
```

MSP430 processor

```
          P1OUT &= ~(BIT2);              // IO1_2 low
```

ATtiny processor

```
          PORTB &= ~0x4;                 // IOB_2 low
```

Blinking the LED

To finish our program, all we need to do is put it all together. The pseudocode for this is:

```
main:
  initialize the direction of the I/O pin to be an output
loop:
  set the LED on
  do nothing for some period of time
  set the LED off
  do nothing for the same period of time
  repeat loop
```

Once you've programmed it for your processor, you should compile, load, and test it. You might want to tweak the delay loop so the LED looks about right. It will probably require several tens of thousands of processor cycles, or the LED will blink faster than you can perceive it.

Troubleshooting

If you have a debugging system such as JTAG set up, finding out why your LED won't turn on is likely to be straightforward. Otherwise, you may have to use the process of elimination.

First, double-check your math. Even if you are completely comfortable with hex and bit shifting, a typo is always possible. In my experience, typos are the most difficult bugs to catch, often harder than memory corruptions. Check you are using the correct pin on the schematic. And make sure there is power to the board. (You may think that advice is funny, but you'd be surprised at how often this plays a role!)

Next, check to see whether a pin is shared between different peripherals. While we said that the pin was an I/O by default, if you are having trouble, check to see if there is an alternate function that may be set in the configuration register.

As long as you have the manual open, verify that the pin is configured properly. If the LED isn't responding, you'll need to read the peripherals chapter of the user manual. Processors are different, so check that the pin doesn't need additional configuration (e.g., a power control output) or have a feature turned on by default (do not make the GPIO an open-drain output by accident).

Most processors have I/O as a default because that is the simplest way for their users (us!) to verify the processor is connected correctly. However, particularly with low-power processors, they want to keep all unused subsystems off to avoid power consumption. (And other special-purpose processors may have other default functionality.) The user manual will tell you more about the default configuration and how to change it. These are often other registers that need a bit set (or cleared) to make a pin act as an I/O.

Sometimes the clocks need to be configured to let the I/O subsystem work correctly (or to make a delay function go slow enough you can see the toggling). If you still have trouble, look for the vendor's examples and identify any differences.

Next, make sure the system is running your code. Do you have another way to verify that the code being run is the code that you compiled? If you have a debug serial port, try incrementing the revision to verify that the code is getting loaded. If those don't work, make sure your build system is using the correct processor for the target.

Make the code as simple as possible to be certain that the processor is running the function handling the LEDs. Eliminate any noncritical

initialization of peripherals in case the system is being delayed while waiting for a nonexistent external device. Turn off interrupts and asserts. Make sure the watchdog is off (see “Watchdog”).

With many microcontrollers, pins can sink more current than they can source (provide). Therefore, it is not uncommon for the pin to be connected to the cathode rather than the anode of the LED. In these instances, you turn on an LED by writing a zero rather than a one. This is called *inverted logic*. Check your schematic to see if this is the case.

If the output still doesn’t work, consider whether there is a hardware issue. If possible, run your sw on multiple boards to rule out an assembly-related defect. Even in hardware, it can be something simple (installing LEDs backward is pretty easy). It may be a design problem such as the processor pin being unable to provide enough current to drive the LED; the datasheet (or user manual) might be able to tell you this. There might be a problem on the board, such as a broken component or connection. With the high-density pins on most processors, it is very easy to short pins together. Ask for help, or get out your multimeter (or oscilloscope).

Separating the Hardware from the Action

Marketing liked your first prototype, though they might want to tweak it a bit later. The system went from a prototype board to a PCB. During this process, somehow the pin number changed (to IO1_3). They need to be able to run both systems.

It is trivially simple to fix the code for this project, but for a larger system, the pins may be scrambled to make way for a new feature. Let’s look at how to make modifications simpler.

Board-Specific Header File

Using a board-specific header file lets you avoid hardcoding the pin. If you have a header file, you just have to change a value there instead of going through your code to change it everywhere it’s referenced. The header file might look like this:

```
#define LED_SET_DIRECTION  (P1DIR)
#define LED_REGISTER       (P1OUT)
#define LED_BIT            (1 << 3)
```

The lines of code to configure and blink the LED can be processor-independent:

```
LED_SET_DIRECTION |= LED_BIT; // set the I/O to be output
LED_REGISTER |= LED_BIT;      // turn the LED on
LED_REGISTER &= ~LED_BIT;     // turn the LED off
```

That could get a bit unwieldy if you have many I/O lines or need the other registers. It might be nice to be able to give only the port (1) and position in the port (3) and let the code figure it out. The code might be more complex, but it is likely to save time (and bugs). For that, the header file would look like this:

```
// ioMapping_v2.h
#define LED_PORT 1
#define LED_PIN 3
```

If we want to recompile to use different builds for different boards, we can use three header files. The first is the old board pin assignments (*ioMapping_v1.h*). Next, we'll create one for the new pin assignment (*ioMapping_v2.h*). We could include the one we need in our main *.c* file, but that defeats the goal of modifying that code less. If we have the main file always include a generic *ioMapping.h*, we can switch the versions in the main file by including the correct header file:

```
// ioMapping.h
#if COMPILING_FOR_V1
#include "ioMapping_v1.h"
#elif COMPILING_FOR_V2
#include "ioMapping_v2.h"
#else
#error "No I/O map selected for the board. What is your target?"
#endif /* COMPILING_FOR_*/
```

Using a board-specific header file hardens your development process against future hardware changes. By sequestering the board-specific information from the functionality of the system, you are creating a more loosely coupled and flexible code base.

NOTE

Keeping the I/O map in Excel is a pretty common way to make sure the hardware and software engineers agree on the pin definitions. With a little bit of creative scripting, you can generate your version-specific I/O map from a CSV file to ensure your pin identifiers match those on the schematic.

I/O-Handling Code

Instead of writing directly to the registers in the code, we'll need to handle the multiple ports in a generic way. So far we need to initialize the pin to be an output, set the pin high so the LED is on, and set the pin low so the LED is off. Oddly enough, we have a large number of options for putting this together, even for such a simple interface.

In the implementation, the initialization function configures the pin to be an output (and sets it to be an I/O pin instead of a peripheral if necessary). With multiple pins, you might be inclined to group all of the initialization together, but that breaks the modularity of the systems.

Although the code will take up a bit more space, it is better to have each subsystem initialize the I/Os it needs. Then if you remove or reuse a module, you have everything you need in one area. However, we've seen one situation where you should *not* separate interfaces into subsystems: the I/O mapping header file, where all of the pins are collected together to make the interface with the hardware more easily communicated.

Moving on with the I/O subsystem interface, setting a pin high and low could be done with one function: `IOWrite(port, pin, high/low)`. Alternatively, this could be broken out so that there are two functions: `IOSet(port, pin)` and `IOClear(port, pin)`. Both methods work. Imagine what our main function will look like in both cases.

The goal is to make the LED toggle. If we use `IOWrite`, we can have a variable that switches between high and low. In the `IOSet` and `IOClear` case, we'd have to save that variable and check it in the main loop to determine which function to call. Alternatively, we could hide `IOSet` and `IOClear` within another function called `IOToggle`. We don't have any particular constraints with our hardware, so we don't need to consider optimizing the code in this example. For education's sake, however, consider the options we are giving ourselves with these potential interfaces.

The `IOWrite` option does everything in one function, so it takes less code space. However, it has more parameters, so it takes more stack space, which comes out of RAM. Plus it has to keep around a state variable (also RAM).

With the `IOSet`/`IOClear`/`IOToggle` option, there are more functions (more code space), but fewer parameters and possibly no required variables (less

RAM). Note that the toggle function is no more expensive in terms of processor cycles than the set and clear functions.

This sort of evaluation requires you to think about the interface along another dimension. Chapter 8 will go over more details on how to optimize for each area. During the prototyping phase, it is too soon to optimize the code, but it is never too soon to consider how the code can be designed to allow for optimization later.

Main Loop

The modifications in the previous sections put the I/O-handling code in its own module, though the basics of the main loop don't change. The implementation might look like the following:

```
void main(void){
  IOSetDir(LED_PORT, LED_PIN, OUTPUT);
  while (1) { // spin forever
    IOToggle(LED_PORT, LED_PIN);
    DelayMs(DELAY_TIME);
  }
}
```

The main function is no longer directly dependent on the processor. With this level of decoupling, the code is more likely to be reused in other projects. In (a) of Figure 4-3, the original version of software architecture is shown, with its only dependency being the processor header. The middle, labeled (b), is our current version. It is more complicated, but the separation of concerns is more apparent. Note that the header files are put off to the side to show that they feed into the dependencies.

Our next reorganization will create an even more flexible and reusable architecture, illustrated by (c).

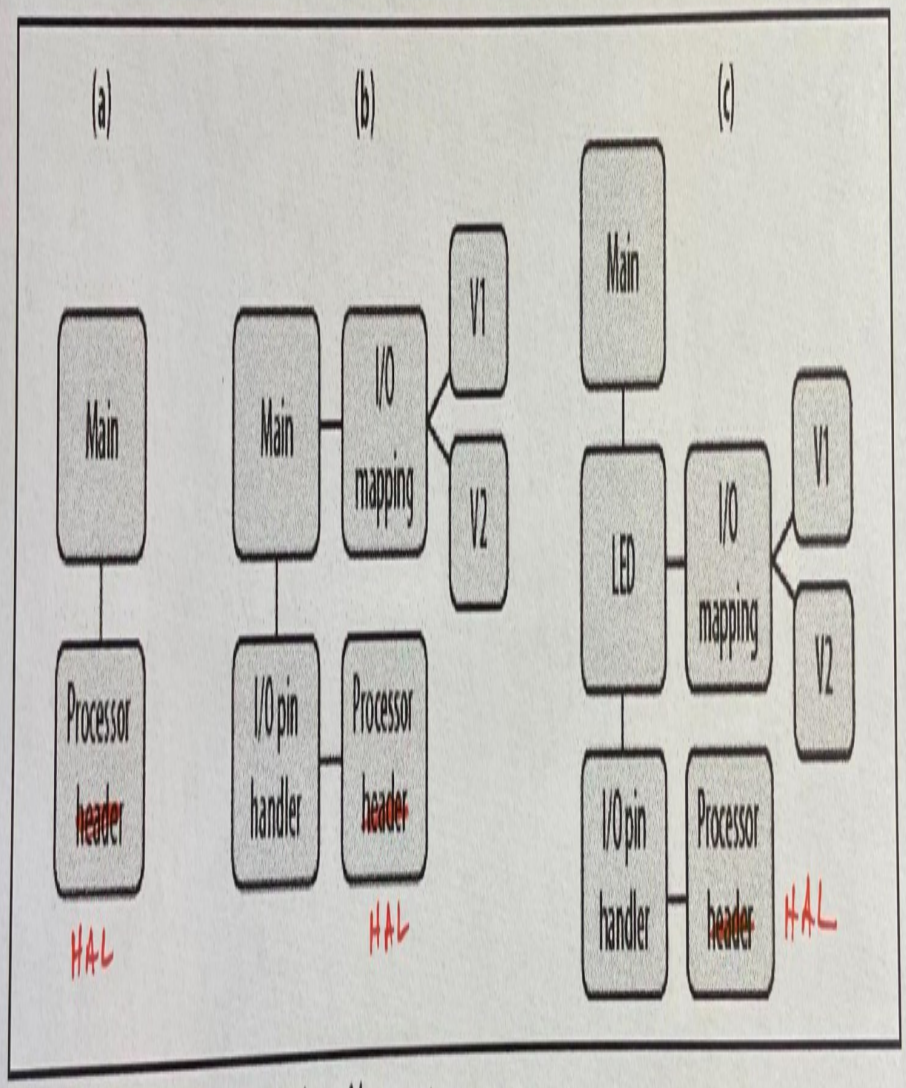


Fig 4-3: Comparison of different Architectures

NOTE TO ARTIST: replace "processor header" with "processor HAL"

Figure 4-3. Comparison of different architectures in order of increasing abstraction

Facade Pattern

As you can imagine, our I/O interface is going to get more complex as the product features expand. (Currently we have only one output pin, so it can't really get any simpler.) In the long run, we want to hide the details of each subsystem. There is a standard software design pattern called *facade* that provides a simplified interface to a piece of code. The goal of the facade pattern is to make a software library easier to use. Along the lines of the metaphor I've been using in this book, that interfacing to the processor is similar to working with a software library, it makes sense to use the facade pattern to hide some details of the processor and the hardware.

In the section "From Diagram to Architecture", we saw the adapter pattern, which is a more general version of the facade pattern. Whereas the adapter pattern acted as a translation between two layers, the facade does this by simplifying the layer below it. If you were acting as an interpreter between scientists and aliens, you might be asked to translate "x=y+2, where y=1." If you were an adapter pattern, you'd restate the same information without any changes. If you were a facade pattern, you'd probably say "x=3" because it is simpler and the details are not critical to using the information.

Hiding details in a subsystem is an important part of good design. It makes the code more readable and more easily tested. Furthermore, the calling code doesn't depend on the internals of the subsystem, so the underlying code can change, while leaving the facade intact.

A facade for our blinking LED would hide the idea of I/O pins from the calling code by creating an LED subsystem, as shown in the right side of Figure 4-3. Given how little the user needs to know about the LED subsystem, the facade could be implemented with only two functions:

`LEDInit()`

: Calls the I/O initialization function for the LED pin (replaces `IOSetDir(…)`)

`LEDBlink` *()*

: Blinks the LED (replaces `IOToggle(…)`)

Adding a facade will often increase the size of your code, but may be worth it in terms of debuggability and maintainability.

The Input in I/O

Marketing has determined that they want to change the way the system blinks in response to a button. Now, when the button is held down, the system should stop blinking altogether.

Our schematic is not much more complex with the addition of a button (see Figure 4-4). Note that the button uses IO2_2 (I/O port 2, pin 2), which is denoted in the schematic with S1 (switch 1). The icon for a switch makes some sense; when you push it in, it conducts across the area indicated. Here, when you press the switch, the pin will be connected to ground.

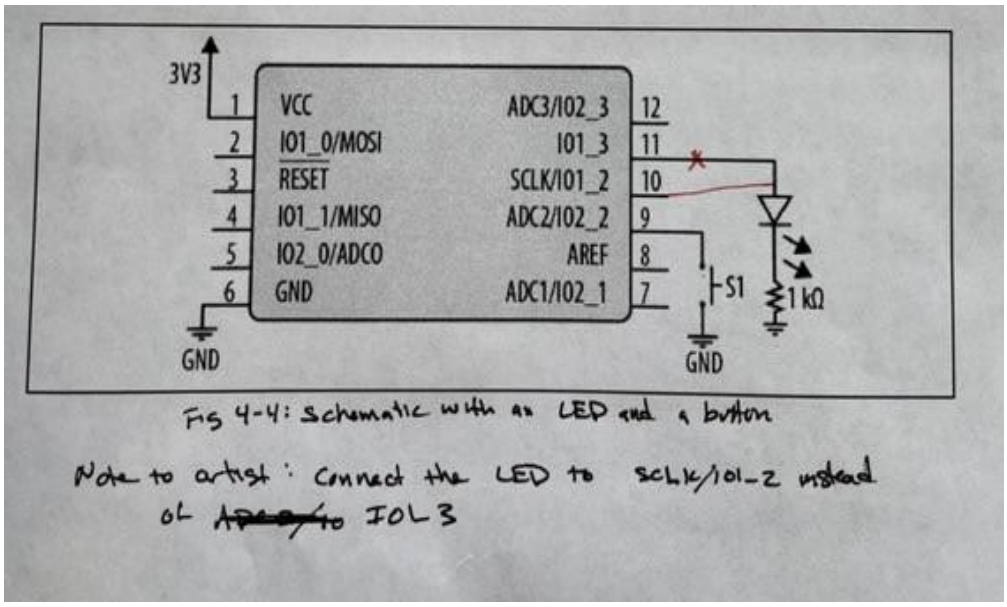


Figure 4-4. Schematic with an LED and a button

Many processor I/O pins have internal pull-up resistors. When a pin is an output, the pull-ups don’t do anything. However, when the pin is an input, the pull-up gives it a consistent value (1), even when nothing is attached. The existence and strength of the pull-up may be settable, but this depends on your processor (and possibly on the particular pin). Most processors even have an option to allow internal pull-downs on a pin. In that case, our switch could have been connected to power instead of ground.

NOTE

> Inputs with internal pull-ups take a bit of power, so if your system needs to conserve a few microamps, you may end up disabling the unneeded pull-ups (or pull-downs).

Your processor user manual will describe the pin options. The basic steps for setup are:

1. Add the pin to the I/O map header file.
2. Configure it to be an input. Verify that it is not part of another peripheral.
3. Configure a pull-up explicitly (if necessary).

Once you have your I/O pin set up as an input, you'll need to add a function to use it, one that can return the state of the pin as high (true) or low (false):

```
boolean IOGet(uint8_t port, uint8_t pin);
```

The button will connect to ground when the button is pressed. This signal is *active low*, meaning that when the button is actively being held down, the signal is low.

A Simple Interface to a Button

To keep the details of the system hidden, we'll want to make a button subsystem that can use our I/O-handling module. On top of the I/O function, we can put another facade so that the button subsystem will have a simple interface.

The I/O function returns the level of the pin. However, we want to know whether the user has taken an action. Instead of the button interface returning the level, you can invert the signal to determine whether the button is currently pressed. The interface could be:

```
void ButtonInit()
```

Calls the I/O initialization function for the button

```
boolean ButtonPressed()
```

Returns true when the button is down

As shown in Figure 4-5, both the LED and button subsystems use the I/O subsystem and I/O map header file. This is a simple illustration of how the modularization we did earlier in the chapter allows reuse.

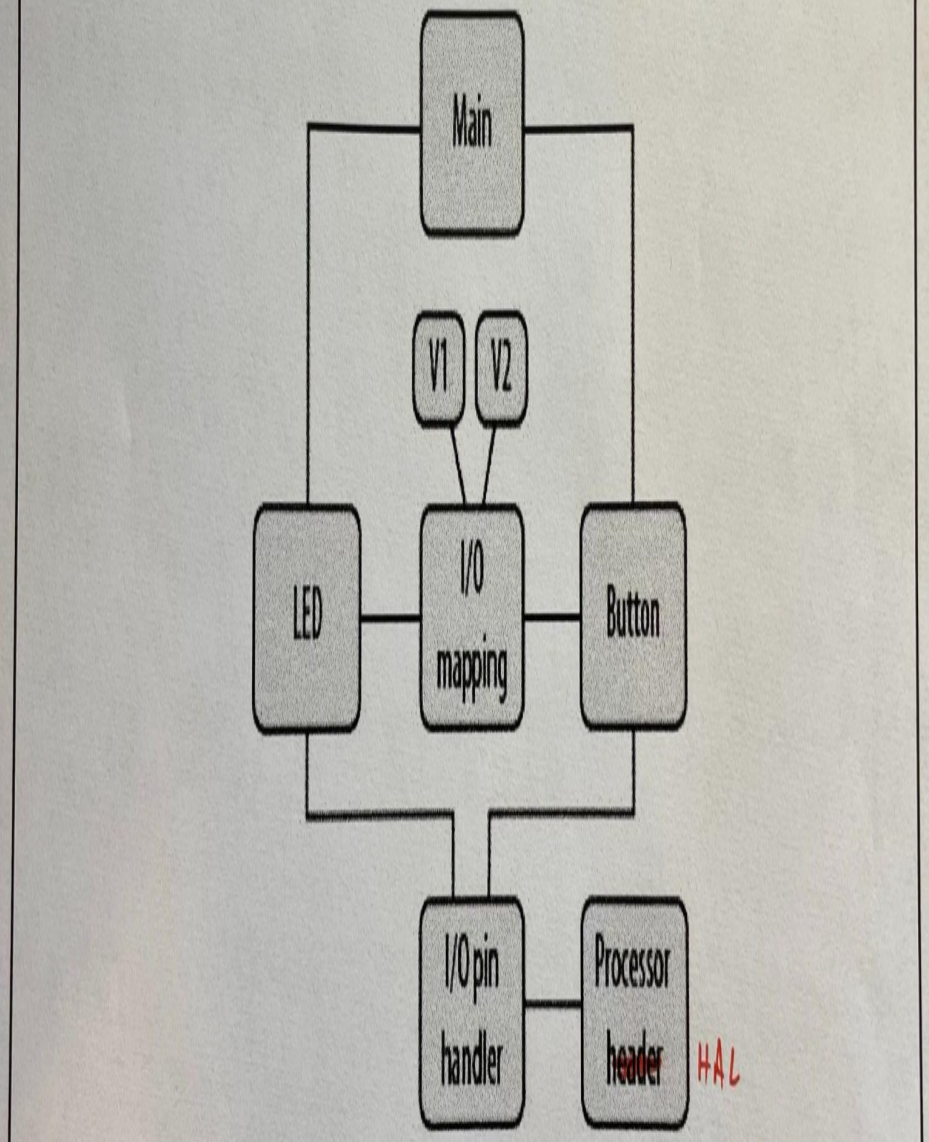


Fig 4-5 Architecture with newly added button

NOTE TO ARTIST: replace "Prosser hand" with "processor HAL"

Figure 4-5. Architecture with the newly added button

At a higher level, there are a few ways to implement the main function. Here is one possibility:

```
main:
  initialize LED
  initialize button
loop:
  if button pressed, turn LED off
  else toggle LED
  do nothing for a period of time
  repeat
```

With this code, the LED won't go off immediately, but will wait until the delay has expired. The user may notice some lag between pushing the button and the LED turning off.

NOTE

A system that doesn't respond to a button press in less than a quarter of a second (250 ms) feels sluggish and difficult to use. A response time of 100 ms is much better, but still noticeable to impatient people. A response time under 50 ms feels very snappy.

To decrease the response time, we could check constantly to see whether the button was pressed:

```
loop:
  if button pressed, turn LED off
  else
    if enough time has passed,
      toggle LED
      clear how much time has passed
  repeat
```

Both of these methods check the button to determine whether it is pressed. This continuous querying is called polling and is easy to follow in the code. However, if the LED needs to be turned off as fast as possible, you may want the button to interrupt the normal flow.

Wait! That word (*interrupt*) is a really important one. I'm sure you've heard it before, and we'll get into a lot more detail soon (and again in Chapter 5). Before that, look at how simple the main loop can be if you use an interrupt to capture and handle the button press:

```
loop:
  if button not pressed, toggle LED
  do nothing for a period of time
  repeat
```

The interrupt code (aka *ISR*, or *interrupt service routine*) calls the function to turn off the LED. However, this makes the button and LED subsystems depend on each other, coupling the systems together in an unobvious way. There are times where you'll have to do this so an embedded system can be fast enough to handle an event.

Chapter 5 will describe how and when to use interrupts in more detail. This chapter will continue to look at them at a high level only.

Momentary Button Press

Instead of using the button to halt the LED, marketing wants to test different blink rates by tapping the button. For each button press, the system should decrease the amount of delay by 50% (until it gets to near zero, at which point it should go back to the initial delay).

In the previous assignment, all you had to check was whether the button was pressed down. This time you have to know both when the button will be pressed and when it will be released. Ideally, we'd like the switch to look like the top part of Figure 4-6. If it did, we could make the system note the rising edge of the signal and take an action there.

Interrupt on a Button Press

This might be another area where an interrupt can help us catch the user input so the main loop doesn't have to poll the I/O pin so quickly. The main loop becomes straightforward if it uses a global variable to learn about the button presses:

```
interrupt when the user presses the button:
  set global button pressed = true

loop:
  if global button pressed,
    set the delay period (reset or decrease it)
    set global button pressed = false
    turn LED off
  if enough time has passed,
    toggle LED
    clear how much time has passed
  repeat
```

The input pins on many processors can be configured to interrupt when the signal at the pin is at a certain level (high or low) or has changed (rising or falling edge). If the button signal looks like it does in the ideal button signal at the top of Figure 4-6, where would you want to interrupt? Interrupting

when the signal is low may lead to multiple activations if the user holds the button down. I prefer to interrupt on the rising edge so that when the user presses the button down, nothing happens until she releases it.

WARNING

To check a global variable accurately in this situation, you'll need the `volatile` C keyword, which perhaps you've never needed before now when developing software in C and C++. The keyword tells the compiler that the value of the variable or object can change unexpectedly and should never be optimized out. All registers and all global variables shared between interrupts and normal code should be marked as volatile. If your code works fine without optimizations and then fails when optimizations are on, check that the appropriate globals and registers are marked as volatile.

Configuring the Interrupt

Configuring a pin to trigger an interrupt is usually separate from configuring the pin as an input. Although both count as initialization steps, we want to keep the interrupt configuration separated from our existing initialization function. This way, you can save the complexity of interrupt configuration for the pins that require it (more in Chapter 5).

Configuring a pin for interrupting the processor adds three more functions to our I/O subsystem:

`IOConfigureInterrupt(`*`port`*`, `*`pin`*`, `*`trigger type`*`, `*`trigger state`*`)`

Configures a pin to be an interrupt. The interrupt will trigger when it sees a certain trigger type, such as edge or level. For an edge trigger type, the interrupt can occur at the rising or falling edge. For a level trigger type, the interrupt can occur when the level is high or low. Some systems also provide a parameter for a callback, which is a function to be called when the interrupt happens; other systems will hardcode the callback to a certain function name, and you'll need to put your code there.

`IOInterruptEnable(`*`port`*`, `*`pin`*`)`

Enables the interrupt associated with a pin.

`IOInterruptDisable(`*`port`*`, `*`pin`*`)`

Disables the interrupt associated with a pin.

If interrupts are not per-pin (they could be per-bank), the processor may have a generic I/O interrupt for each bank, in which case the interrupt service routine (ISR) will need to untangle which pin caused the interrupt. It depends on your processor. If each I/O pin can have its own interrupt, the modules can be more loosely coupled.

Debouncing Switches

Many buttons do not provide the clean signal shown in the ideal button signal in the top part of Figure 4-6. Instead, they look more like those labeled "Bouncy digital button signal." If you interrupted on that signal, your system could waste processor cycles by interrupting on the glitches at the start and end of the button press. *Switch bouncing* can be due to a mechanical or electrical effect.

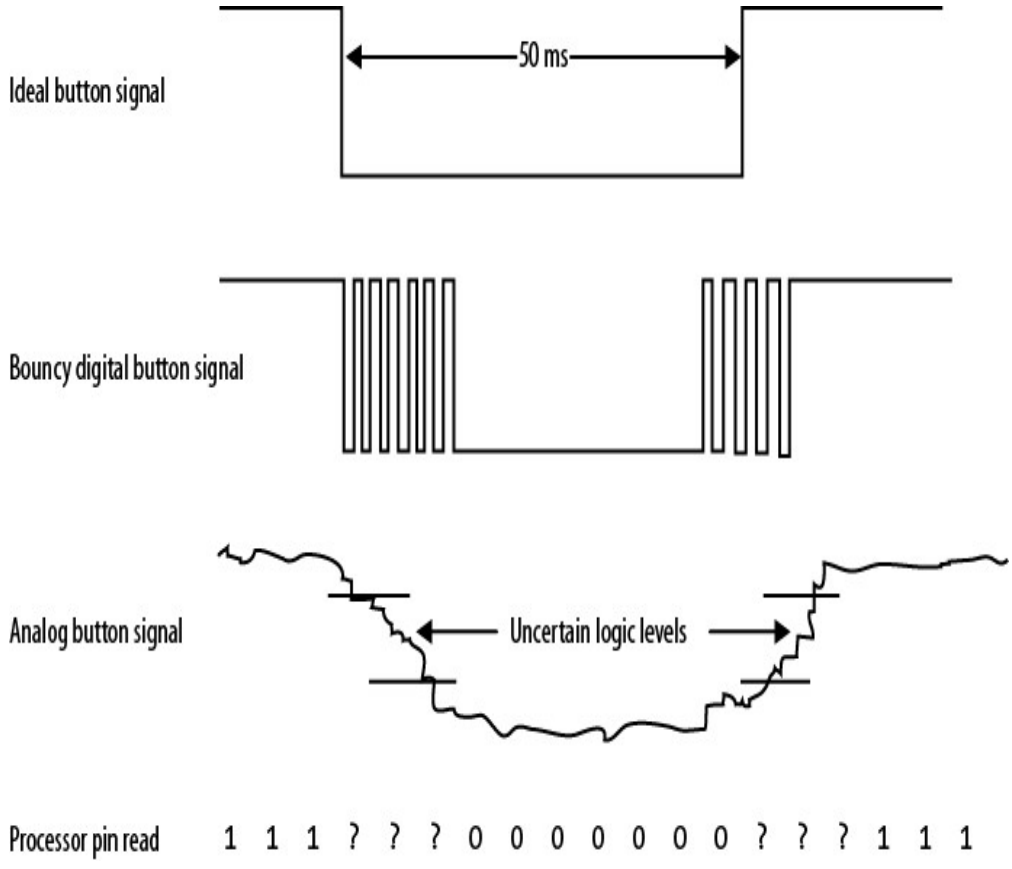


Figure 4-6. Different views of button signals

Figure 4-6 shows an analog view of what could happen when a button is pressed and only slowly goes into effect. (What really happens can be a lot more complicated, depending on whether the bouncing is due primarily to a

mechanical or electrical cause.) Note that there are parts of the analog signal where the signal is neither high nor low, but somewhere in between. Since your I/O line is a digital signal, it can't represent this indeterminate value and can behave badly. A digital signal full of edges would cause the code to believe multiple presses happened per user action. The result would be inconsistent and frustrating to the user. Worse, interrupting on such a signal can lead to processor instability, so let's go back to polling the signal.

Debouncing is the technique used to eliminate the spurious edges. Although it can be done in hardware or software, we'll focus on software. See Jack Ganssle's excellent web article on debouncing (in the section "Further Reading") for hardware solutions.

NOTE

Many modern switches have a very short period of uncertainty. Switches have datasheets, too; check yours to see what the manufacturer recommends. Beware that trying to empirically determine whether you need debouncing might not be enough, as different batches of switches may act differently.

You still want to look for the rising edge of the signal, where the user releases the button. To avoid the garbage near the rising and falling edges, you'll need to look for a relatively long period of consistent signal. How long that is depends on your switch and how fast you want to respond to the user.

To debounce the switch, take multiple readings (aka samples) of the pin at a periodic interval several times faster than you'd like to respond. When there have been several consecutive, consistent samples, alert the rest of the system that the button has changed. See Figure 4-6; the goal is to read enough that the uncertain logic levels don't cause spurious button presses.

You will need three variables:

- The current raw reading of the I/O line
- A counter to determine how long the raw reading has been consistent
- The debounced button value used by the rest of the code

How long debouncing takes (and how long your system takes to respond to a user) depends on how high the counter needs to increment before the debounced button variable is changed to the current raw state. The counter should be set so that the debouncing occurs over a reasonable time for that switch.

If there isn't a specification for it in your product, consider how fast buttons are pressed on a keyboard. If advanced typists can type 120 words per minute, assuming an average of five characters per word, they are hitting keys (buttons) about 10 times per second. Figuring that a button is down half the time, you need to look for the button to be down for about 50 ms. (If you really are making a keyboard, you probably need a tighter tolerance because there are faster typists.)

For our system, the mythical switch has an imaginary datasheet stating that the switch will ring for no more than 12.5 ms when pressed or released. If the goal is to respond to a button held down for 50 ms or more, we can sample at 10 ms (100 Hz) and look for five consecutive samples.

Using five consecutive samples is pretty conservative. You may want to adjust how often you poll the pin's level so you need only three consecutive samples to indicate that the button state has changed.

NOTE

Strike a balance in your debouncing methodology: consider the cost of being wrong (annoyance or catastrophe?) and the cost of responding slower to the user.

In the previous falling-edge interrupt method of handling the button press, the state of the button wasn't as interesting as the change in state. To that end, we'll add a fourth variable to simplify the main loop.

```
read button:
  if raw reading same as debounced button value,
    reset the counter
  else
    decrement the counter
    if the counter is zero,
      set debounced button value to raw reading
      set changed to true
      reset the counter

main loop:
  if time to read button,
    read button
```

```
    if button changed and button is no longer pressed
      set button changed to false
      set the delay period (reset or halve it)
  if time to toggle the LED,
    toggle LED
  repeat
```

Note that this is the basic form of the code. There are many options that can depend on your system's requirements. For example, do you want it to react quickly to a button press but slowly to a release? There is no reason the debounce counter needs to be symmetric.

In the preceding pseudocode, the main loop polls the button again instead of using interrupts. However, many processors have timers that can be configured to interrupt. Reading the button could be done in a timer to simplify the main function. The LED toggling could also happen in a timer. More on timers soon, but first, marketing has another request.

Runtime Uncertainty

Marketing has a number of LEDs to try out. The LEDs are attached to different pins. Use the button to cycle through the possibilities.

We've handled the button press, but the LED subsystem knows only about the output on pin 1_2 on the v1 board or 1_3 on the v2 board. Once you've initialized all the LEDs as outputs, you could put a conditional (or switch) statement in your main loop:

```
if count button presses == 0, toggle blue LED
if count button presses == 1, toggle red LED
if count button presses == 2, toggle yellowbLED
```

To implement this, you'll need to have three different LED subsystems or (more likely) your LED toggle function will need to take a parameter. The former represents a lot of copied code (almost always a bad thing), whereas the latter means the LED function will need to map the color to the I/O pin each time it toggles the LED (which consumes processor cycles).

Our goal here is to create a method to use one particular option from a list of several possible objects. Instead of making the selection each time (in the main loop or in the LED function), you can select the desired LED when the button is pressed. Then the LED toggle function is agnostic about which LED it is changing:

```
main loop:
  if time to read button,
    read button
    if button changed and button is no longer pressed
```

```
    set button changed to false
    change LED variable

if time to toggle the LED,
  toggle LED
repeat
```

By adding a state variable, we use a little RAM to save a few processor cycles. State variables tend to make a system confusing, especially when the `change LED variable` section of the code is separated from `toggle LED`. Unraveling the code to show how a state variable controls the option can be tedious for someone trying to fix a bug (commenting helps!). However, the state variable simplifies the LED toggle function considerably, so there are times where a state variable is worth the complications it creates.

Dependency Injection

However, we can go beyond a state variable to something even more flexible. Earlier we saw that abstracting the I/O pins from the board saves us from having to rewrite code when the board changes. We can also use abstraction to deal with dynamic changes (such as which LED is to be used). For this, we'll use a technique called *dependency injection*.

Before, we were hiding the I/O pin in the LED code (creating a hierarchy of functions that depend only on the lower levels). With dependency injection, we'll remove that dependency by passing an I/O handler as a parameter to the LED initialization code. The I/O handler will know which pin to change and how to change it, but the LED code will know only how to call the I/O handler. See Figure 4-7.

An oft-used example to illustrate dependency injection relates engines to cars. The car, the final product, depends on an engine to move. The car and engine are made by the manufacturer. Though the car cannot choose which engine to install, the manufacturer can inject any of the dependency options that the car can use to get around (e.g., the 800-horsepower engine or the 20-horsepower one).

Tracing back to the LED example, the LED code is like the car and depends on the I/O pin to work, just as the car depends on the engine. However, the LED code may be made generic enough to avoid dependence on a particular I/O pin. This allows the main function (our manufacturer) to install an I/O pin appropriate to the circumstance instead of hardcoding the dependency at

compile time. This technique allows you to compose the way the system works at runtime.

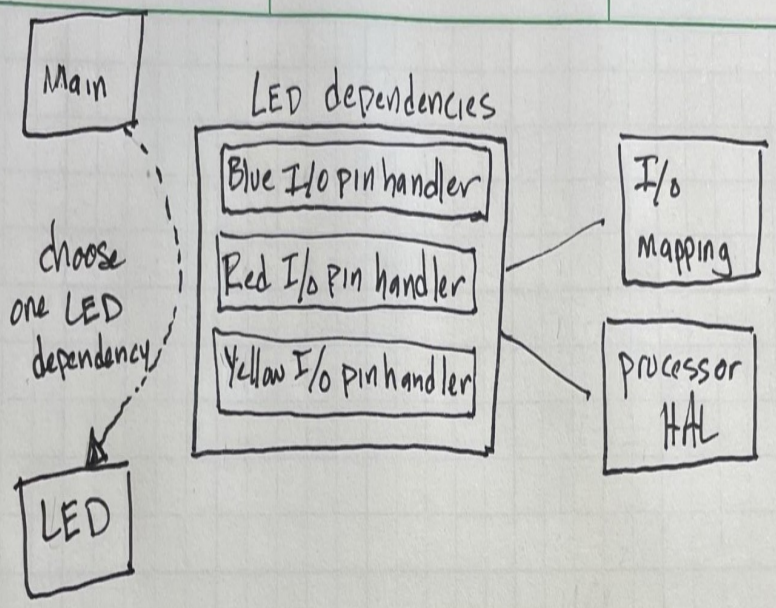


Figure 6-b: Architecture showing dependency injection

NOTE TO ARTIST

The Main passes one of the LED dependencies into the LED code (via a dashed line and a note saying "Choose one LED dependency"). There are three possible LEDs: blue, red and yellow. Each handles the pins neccessary to change the desired LED. The dependencies use the I/o Mapping and processor HAL components.

This should be in the style used for Fig 4-3 and Fig 4-5.

Figure 4-7. Dependency injection architecture for runtime uncertainty

In C++ or other object-oriented languages, to inject the dependency, we pass a new I/O pin handler object to the LED whenever a button is pressed. The LED module would never know anything about which pin it was changing or how it was doing so. The variables to hide this are set at initialization time (but do remember that these are variables, consuming RAM and cluttering the code).

NOTE

A structure of function pointers is often used in C to achieve the same goal.

Dependency injection is a very powerful technique, particularly if your LED module were to do something a lot more complicated, for instance, output Morse code. If you passed in your I/O pin handler, you could reuse the Morse code LED output routine for any processor. Further, during testing, your I/O pin handler could print out every call that the LED module made to it instead of (or in addition to) changing the output pin.

However, the car engine example illustrates one of the major problems with dependency injection: complexity. It works fine when you only need to change the engine. But once you are injecting the wheels, the steering column, the seat covers, the transmission, the dashboard, and the body, the car module becomes quite complicated, with little intrinsic utility of its own.

The aim of dependency injection is to allow flexibility. This runs contrary to the goal of the facade pattern, which reduces complexity. In an embedded system, dependency injection will take more RAM and a few extra processor cycles. The facade pattern will almost always take more code space. You will need to consider the needs and resources of your system to find a reasonable balance.

Using a Timer

Using the button to change the blinking speed was helpful, but marketing has found a problem in the uncertainty introduced into the blink rate. Instead of cutting the speed of the LED in half, they want to use the button to cycle

through a series of precise blink rates: 6.5 times per second (Hz), 8.5 Hz, and 10 Hz.

This request seems simple, but it is the first time we've needed to do anything with time precision. Before, the system could handle buttons and toggle the LED generally when it was convenient. Now the system needs to handle the LED in real time. How close you get to "precise" depends on the parameters of your system, mainly on the accuracy and precision of your processor input clock. We'll start by using a timer on the processor to make it more precise than it was before, and then see if marketing can accept that.

Timer Pieces

In principle, a timer is a simple counter measuring time by accumulating a number of clock ticks. The more deterministic the master clock is, the more precise the timer can be. Timers operate independently of software execution, acting in the background without slowing down the code at all. To be technical, they are happening in the silicon gates that are part of the microcontroller.

To set the frequency of the timer, you will need to determine the clock input. This may be your processor clock (aka the *system clock*, or *master clock*), or it may be a different clock from another subsystem (for instance, many processors have a peripheral clock).

SYSTEM STATISTICS

When embedded systems engineers talk about the stats of our systems to other engineers, we tend to use a shorthand consisting of the vendor, the processor (and its core), the number of bits in each instruction, and our system clock speed. Earlier in this chapter, I gave register examples from the STM32F103xx, MSP430, and ATtiny processor families. Systems with these processors could have stats like this:

- STM32F103 (Cortex-M3), 32-bit, 72 MHz
- Texas Instrument MSP430 G2201, 16-bit, 16 MHz
- Atmel ATtiny45, 8-bit, 4 MHz

That last number is the processor clock, which describes the theoretical number of instructions the processor can handle in a second. The actual performance may be slower if your memory accesses can't keep up, or faster if you can use processor features to bypass overhead. The system clock is not the same as the oscillator on the board (if you have one). Thanks to an internal PLL circuit, your processor speed might be much faster than an onboard oscillator. PLL stands for *phase lock loop* and denotes a way that a processor can multiply a slower clock (i.e., a slow oscillator) to get a faster clock (a processor clock). Because slower oscillators are generally cheaper (and consume less power) than faster ones, PLLs are ubiquitous.

Many small microcontrollers use an internal RC oscillator as their clock source. Although these make life easier for the hardware designer, their accuracy leaves a lot to be desired. Considerable drift can accumulate over time, and this can lead to errors in communication and some real-time applications.

For example, the ATtiny45 has a maximum processor clock of 4 MHz. We want the LED to be able to blink at 10 Hz, but that means interrupting at 20 Hz (interrupt to turn it on, interrupt again to turn it off). This means we'll need a division of 200,000. The ATtiny45 is an 8-bit processor; it has two 8-bit timers and a 16-bit timer. Neither size of timer will work to count up that high (see the sidebar "System Statistics"). However, the chip designers recognized this issue and gave us another tool: the *prescaler register*, which divides the clock so that the counter increments at a slower rate.

WARNING

Many timers are zero-based instead of one-based so for a prescaler that divides by 2, you put a 1 in the prescaler register. This whole timer thing is complicated enough without carrying it through the math above. Look in your processor manual to see which timer registers are zero-based..

The effect of the prescaler register is seen in Figure 4-8. The system clock toggles regularly. With a prescaler value of two, the prescaled clock (the input to our timer subsystem) toggles at half the system clock speed. The timer counts up. The processor notes when the timer matches the *compare*

register (set to 3 in the diagram). When the timer matches, it may either continue counting up or reset. This behavior depends on the processor and the configuration settings.

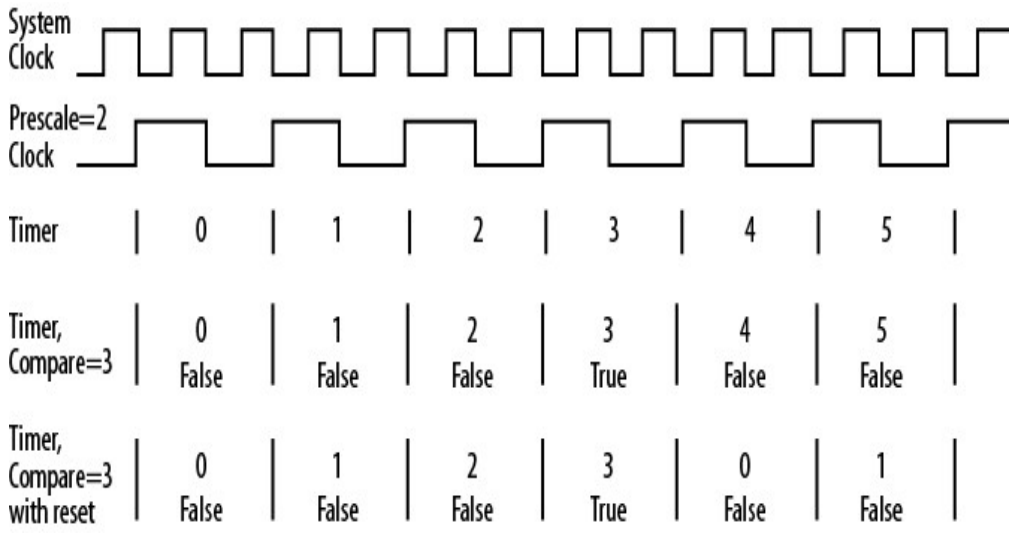


Figure 4-8. Timer prescaling and matching

Before getting back to the timer on the ATtiny45, note that the registers needed to make a timer work generally consist of the following:

Timer counter

This holds the changing value of the timer (the number of ticks since the timer was last reset).

Compare register (or capture compare register or match register)

When the timer counter equals this register, an action is taken. There may be more than one compare register for each timer.

Action register (or auto reload register)

This register sets up an action to take when the timer and compare register are the same. (For some timers, these actions are also available when the timer overflows, which is like having a compare register set to the maximum value of the timer counter.) There are four types of possible actions to be configured (one or more may happen):

- Interrupt
- Stop or continue counting

- Reload the counter
- Set an output pin to high, low, toggle, or no change

Clock configure register (optional)

This register tells a subsystem which clock source to use, though the default may be the system clock. Some processors have timers that even allow a clock to be attached to an input pin. You can often choose whether to count up or down, I'll be using count-up with these examples but the process is similar.

Prescaler register

As shown in Figure 4-7, this reduces the fast incoming clock so that it runs more slowly, allowing timers to happen for slower events.

Control register

This sets the timer to start counting once it has been configured. Often the control register also has a way to reset the timer.

Interrupt register (may be multiple)

If you have timer interrupts, you will need to use the appropriate interrupt register to enable, clear, and check the status of each timer interrupt.

Setting up a timer is processor-specific, and the user manual generally will guide you through setting up each of these registers. Your processor user manual may give the registers slightly different names.

WARNING

Instead of a compare register, your processor might allow you to trigger the timer actions only when the timer overflows. This is an implicit match value of two to the number of bits in the timer register minus 1 (e.g., for an 8-bit timer, $(2^8)–1 = 255$). By tweaking the prescaler, most timer values are achievable without too much error.

Doing the Math

If you aren't setting up a timer right now, what follows is a lot of math. Feel free to skim or skip ahead to "Pulse Width Modulation." Come back when you need it. Also, check out this book's repository for some calculators that let you go through this math without following my algebra here.

Timers are made to deal with physical timescales, so you need to relate a series of registers to an actual time. Remember that the frequency (for instance, 10 Hz) is inversely proportional to the period (0.1 seconds).

The basic equation for the relationship between the interrupt frequency, clock input, prescaler, and compare register is:

```
interruptFrequency = clockIn / (prescaler * compare)
```

This is an optimization problem. You know the `clockIn` and the goal, `interruptFrequency`. You need to adjust the prescaler and compare registers until the interrupt frequency is close enough to the goal. If there were no other limitations, this would be an easy problem to solve (but it isn't always so easy).

HOW MANY BITS IN THAT NUMBER?

Better than counting sheep, figuring out powers of two is often how I fall asleep. If you don't share the habit, there are some powers of two that you really should memorize.

The number of different values a variable can have is 2 to the power of the number of bits it has (e.g., 8 bits offers 2^8, so 256 different values can be in an 8-bit variable). However, that number has to hold a 0 as well, so the maximum value of an 8-bit variable is $2^8–1$, or 255. Even that is true only for unsigned variables. A signed variable uses one bit for the sign (+/–), so an 8-bit variable would have only seven bits available for the value. Its maximum would be $2^7–1$, or 127. Zero has to be represented only once, so the minimum value of a signed 8-bit variable is –128.

Bits	Power of two	Maximum value	Significance
4	$2^4 = 16$	15	A nibble.

7	2^7 = 128	127	Signed 8-bit variable.
8	2^8 = 256	255	Byte and an unsigned 8-bit variable.
10	2^{10} = 1024	1023	Many peripherals are 10-bit.
12	2^{12} = 4096	4095	Many peripherals are 12-bit.
15	2^{15} = 32768	32767	Signed 16-bit variable.
16	2^{16} = 65536	65535	Unsigned 16-bit variable.
24	2^{24} = 16777216	~1.6 million	Color is often 24-bit.
31	2^{31} = 2147483648	~2 billion	Signed 32-bit variable.
32	2^{32} = 4294967296	~4 billion	Unsigned 32-bit variable.

> I usually fall asleep before 2^{20}, so I remember the higher-order numbers as estimates only. There will be times when a 32-bit number is too small to hold the information you need. Even a 64-bit number can fall down when you build a machine to do something as seemingly simple as shuffling cards.
>
> Remember that variables (and registers) have sizes and that those sizes matter.

Returning to the ATtiny45's 8-bit timer, 4 MHz system clock, and goal frequency of 20 Hz, we can export the constraints we'll need to use to solve the equation:

- These are integer values, so the prescaler and compare register have to be whole numbers. This constraint is true for any processor.
- The compare register has to lie between 0 and 255 (because the timer register is eight bits in size).
- The prescaler on the ATtiny45 is 10 bits, so the maximum prescaler is 1023. (The size of your prescaler may be

different.)

The prescaler for this subsystem is not shared with other peripherals, so we don’t need to be concerned about this potential constraint for our solution (yet).

There are several heuristics for finding a prescaler and compare register that will provide the interrupt frequency needed (see Figure 4-9).

NOTE

I asked two math professors how to solve this problem in a generic manner. The answers I got back were interesting. The most interesting part was learning that this problem is NP-complete for two reasons: integers are involved, and it is a nonlinear, two-variable problem. Thanks, Professor Ross and Professor Patton!

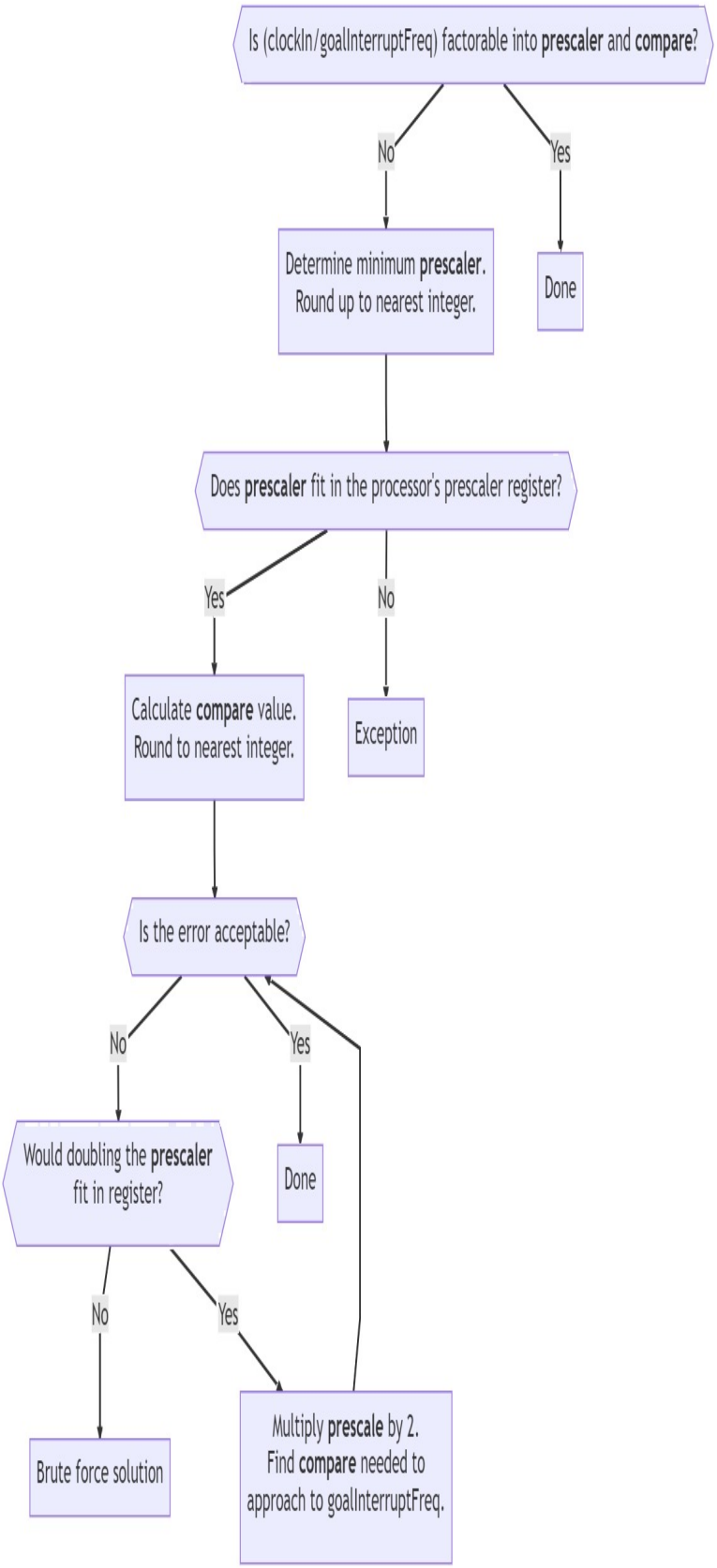

Is (clockIn/goalInterruptFreq) factorable into **prescaler** and **compare**?
No
Yes
Determine minimum **prescaler**. Round up to nearest integer.
Done
Does **prescaler** fit in the processor's prescaler register?
Yes
No
Calculate **compare** value. Round to nearest integer.
Exception
Is the error acceptable?
No
Yes
Would doubling the **prescaler** fit in register?
Done
No
Yes
Brute force solution
Multiply **prescale** by 2. Find **compare** needed to approach to goalInterruptFreq.

Figure 4-9. Heuristic for finding the register values for a goal interrupt frequency

We can determine the minimum prescaler by rearranging the equation and setting the compare register to its maximum value:

```
prescaler = clockIn / (compare * interruptFrequency)
      = 4 MHz / (255 * 20 Hz)
```

Unfortunately, the resulting prescaler value is a floating-point number (784.31). If you round up (785), we end up with an interrupt frequency of 19.98 Hz, which is an error of less than a tenth of a percent off.

If you round down on the prescaler (784), the interrupt frequency will be above the goal and you may be able to decrease the compare register to get the timer to be about right. With prescaler = 784 and compare = 255, the error is 0.04%. However, marketing asked for high precision, and there are some methods to find a better prescaler.

First, note that you want the product of the prescaler and compare register to equal the clock input divided by the goal frequency:

```
prescaler * compare = clockIn / interruptFrequency = 4 MHz/20 Hz = 200,000
```

This is a nice, round number, easily factored into 1,000 (prescaler) and 200 (compare register). This is the best and easiest solution to optimizing the `prescaler` and `compare`: determine the factors of `(clockIn/interruptFrequency)` and arrange them into `prescaler` and `compare`. However, this requires the `(clockIn/interruptFrequency)` to be an integer and the factors to split easily into the sizes allowed for the registers. It isn’t always possible to use this method.

More Math: Difficult Goal Frequency

As we move along to another blink frequency requested by marketing (8.5 Hz, or an interrupt frequency of 17 Hz), the new target is:

```
prescaler * compare = clockIn / interruptFrequency = 4 MHz/17 Hz = 235294.1
```

There is no simple factorization of this floating-point number. We can verify that a result is possible by calculating the minimum prescaler (we did that earlier by setting the compare register to its maximum value). The result (923) will fit in our 10-bit register. We can calculate the percent error using the following:

```
error = 100 * (goal interrupt frequency - actual) / goal
```

With the minimum prescaler, we get an error of 0.03%. This is pretty close, but we may be able to get closer.

Set the prescaler to its maximum value, and see what the options are. In this case, a prescaler of 1,023 leads to a compare value of 230 and an error of less than 0.02%, a little better. But can we reduce the error further?

For larger timers, you might try a binary search for a good value, starting out with the minimum prescaler. Double it. Look at the prescaler values that are +/– 1 to find a compare register that is the closest whole number. If the resulting frequency is not close enough, repeat the doubling of the modified prescaler. Unfortunately, with our example, we can't double our prescaler and stay within the bounds of the 10-bit number.

Finally, another way to find the solution is to use a script or program (e.g., Matlab or Excel) and use brute force to try out the options, as shown in the procedure below. Start by finding the minimum prescaler value and the maximum prescaler value (by setting the compare register to 1). Limit the minimum and maximum so they are integers and fit into the correct number of bits. Then, for each whole number in that range, calculate the compare register for the goal interrupt frequency. Round the compare register to the nearest whole number, and calculate the actual interrupt frequency. This method led to a prescaler of 997, a compare register of 236, and a tiny error of 0.0009%. A brute-force solution like this will give you the smallest error, but will probably take the most developer time. Determine what error you can live with, and go on to other things once you've met that goal.

Brute force method for finding the lowest error interrupt frequency register value:

1. Calculate the minimum and maximum prescaler values:

$$minPrescaler = \frac{clockInput}{goalFrequency \times maxCompare}$$
$$= \frac{4MHz}{17\,Hz \times 255} = 922.722$$
$$maxPrescaler = \frac{clockInput}{goalFrequency \times maxCompare}$$
$$= \frac{4MHz}{7\,Hz \times 255} = 235294$$
$$= 2^{10} - 1 = 1023$$

2. For each prescaler value from 922 to 1023, calculate a compare value:

$$compare = \frac{clockInput}{goalFrequency \times prescaler} = \frac{4MHz}{20Hz \times 922} = 255.2$$
$$= round(255.2) \Rightarrow 255$$

3. Use the calculated compare value to determine the interrupt frequency with these values:

$$interruptFrequency = \frac{clockInput}{prescaler \times compare} = \frac{4MHz}{923 \times 255} = 17.013\,Hz$$

4. Calculate the error:

$$error\,\% = 100 * \frac{abs(goalFrequency - interruptFrequency)}{goalFrequency}$$
$$= 100 * \frac{abs(17\,Hz - 17.013\,Hz)}{17\,Hz} = 0.0783$$

5. Find the prescaler and compare registers with the least error.

A Long Wait Between Timer Ticks

Brute force works well for 17 Hz, but when you get the goal output of 13 Hz, the minimum prescaler that you calculate is more than 10 bits. The timer

cannot fit in the 8-bit timer. This is shown as an exception in the flowchart (Figure 4-8). The simplest solution is to use a larger timer if you can. The ATtiny45's 16-bit timer can alleviate this problem because its maximum compare value is 65535 instead of the 8-bit 255, so we can use a smaller prescaler.

If a larger timer is unavailable, another solution is to disconnect the I/O line from the timer and call an interrupt when the timer expires. The interrupt can increment a variable and take action when the variable is large enough. For example, to get to 13 Hz, we could have a 26 Hz timer and toggle the LED every other time the interrupt is called. This method is less precise because there could be delays due to other interrupts.

Using a Timer

Once you have determined your settings, the hard part is over, but there are a few more things to do:

- Remove the code in the main function to toggle the LED. Now the main loop will need only a set of prescaler and compare registers to cycle through when the button is pressed.
- Write the interrupt handler that toggles the LED.
- Configure the pin. Some processors will connect any timer to any output, whereas others will allow a timer to change a pin only with special settings. For the processor that doesn't support a timer on the pin, you will need to have an interrupt handler in the code that toggles only the pin of interest.
- Configure timer settings and start the timer.

Using Pulse Width Modulation

Market research has shown that potential customers are bothered by the brightness of the LED, saying its laser-like intensity is blinding them. Marketing wants to try out different brightness settings (100%, 80%, 70%, and 50%), using the button to switch between the options.

This assignment gives us a fine opportunity to explore *pulse-width modulation* (PWM), which determines how long a pin stays high or low.

PWMs operate continuously, turning a peripheral on and then off on a regular schedule. The cycle is usually very fast, on the order of milliseconds (or hundreds of Hz).

PWM signals often drive motors and LEDs (though motors require a bit more hardware support). Using PWM, the processor can control the amount of power the hardware gets. Using some inexpensive electronics, the output of a PWM pin can be smoothed to be the average signal. For LEDs, though, no additional electronics are necessary. The brightness is relative to the amount of time the LED is on during each cycle.

A timer is a set of pulses that are all alike, so the timer signal is 50% high and 50% low (this is known as 50% *duty cycle*). In PWM, the pulses' widths change depending on the situation. So a PWM can have a different ratio. A PWM with a 100% duty cycle is always on, like a high level of an output pin. And a 0% duty cycle represents a pin that has been driven (or pulled) low. The duty cycle represents the average value of the signal, as shown by the dashed line in Figure 4-10.

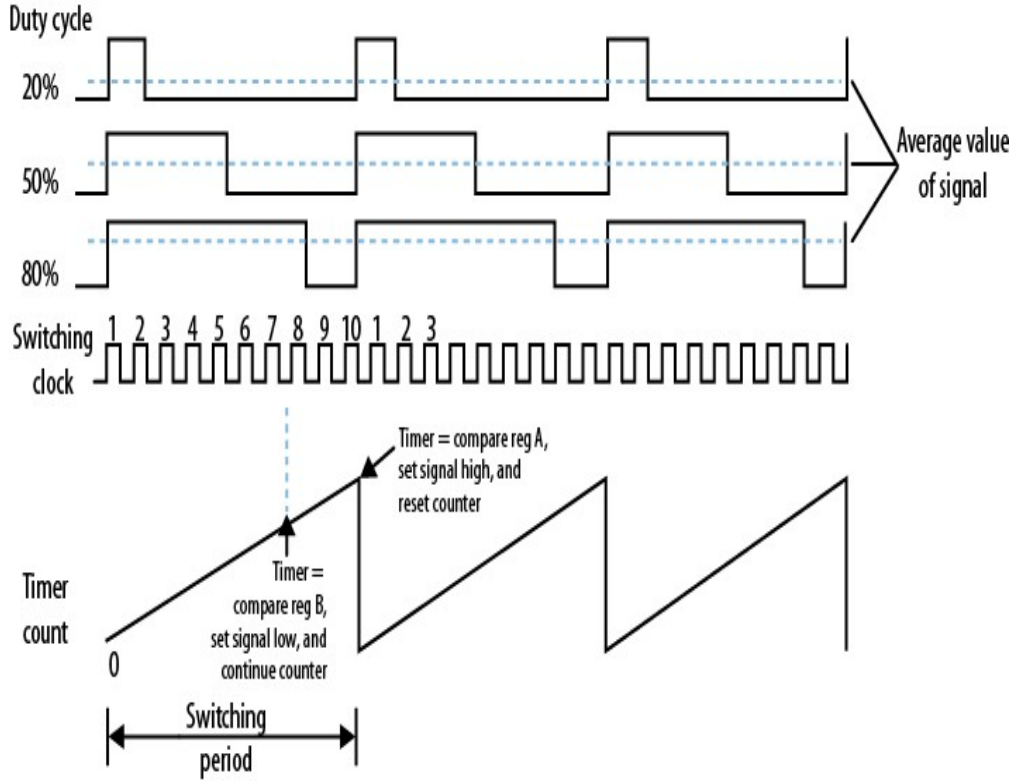


Figure 4-10. PWM duty cycles: 20%, 50% ,and 80%

Given the timer from the previous chapter, we could implement a PWM with an interrupt. For our 20 Hz LED example, we had a compare register of 200

so that every 200 ticks, the timer interrupt would do something (toggle the LED). If we wanted the LED to be on 80% of the time with a 20 Hz timer (instead of the 50% we were doing earlier), we could ping-pong between two interrupts that would set the compare register at every pass:

- Timer interrupt 1
    1. Turn on LED.
    2. Set the compare register to 160 (80% of 200).
    3. Reset the timer.
- Timer interrupt 2
    1. Turn LED off.
    2. Set the compare register to 40 (20% of 200).
    3. Reset the timer.

With a 20 Hz timer, this would probably look like a very quick series of flashes instead of a dim LED. The problem is that the 20 Hz timer is too slow. The more you increase the frequency, the more the LED will look dim instead of blinking. However, a faster frequency means more interrupts.

There is a way to carry out this procedure in the processor. In the previous section, the configurable actions included whether to reset the counter as well as how to set the pin. Many timers have multiple compare registers and allow different actions for each compare register. Thus, a PWM output can be set with two compare registers, one to control the switching frequency and one to control the duty cycle.

For example, the bottom of Figure 4-10 shows a timer counting up and being reset. This represents the *switching frequency* set by a compare register. We'll name this compare register A and set it to 100. When this value is reached, the timer is reset and the LED is turned on. The duty cycle is set with a different register (compare register B, set to 80) that turns the LED off but allows the timer to continue counting.

Which pins can act as PWM outputs depends on your processor, though often they are a subset of the pins that can act as timer outputs. The PWM section

of the processor user manual may be separate from the timer section. Also, there are different PWM controller configurations, often for particular applications (motors are often finicky about which type of PWM they require).

For our LED, once the PWM is set up, the code only needs to modify the duty cycle when the button is pressed. As with timers, the main function doesn't control the LED directly at all.

Although dimming the LED is what marketing requested, there are other neat applications you can try out. To get a snoring effect, where the LED fades in and out, you'll need to modify the duty cycle in increments. If you have tricolor LEDs, you can use PWM control to set the three LED colors to different levels, providing a whole palette of options.

Shipping the Product

Marketing has found the perfect LED (blue), blink rate (8 Hz), and brightness (100%). They are ready to ship the product as soon as you set the parameters.

It all seems so simple to just set the parameters and ship the code. However, what does the code look like right now? Did the timer code get morphed into the PWM code, or is the timer interrupt still around? With a brightness of 100%, the PWM code isn't needed any longer. In fact, the button code can go. The ability to choose an LED at runtime is no longer needed. The old board layout can be forgotten in the future. Before shipping the code and freezing development, let's try to reduce the spaghetti into something less tangled.

A product starts out as an idea and often takes a few iterations to solidify into reality. Engineers often have good imaginations for how things will work. Not everyone is so lucky, so a good prototype can go a long way toward defining the goal.

However, keeping around unneeded code clutters the code base (see the left side of Figure 4-11). Unused code (or worse, code that has been commented out) is frustrating for the next person who doesn't know why things were removed. Avoid that as much as possible. Instead, trust that your version control system can recover old code. Don't be afraid to tag, branch, or release a development version internally. It will help you find the removed features after you prune the code for shipment.

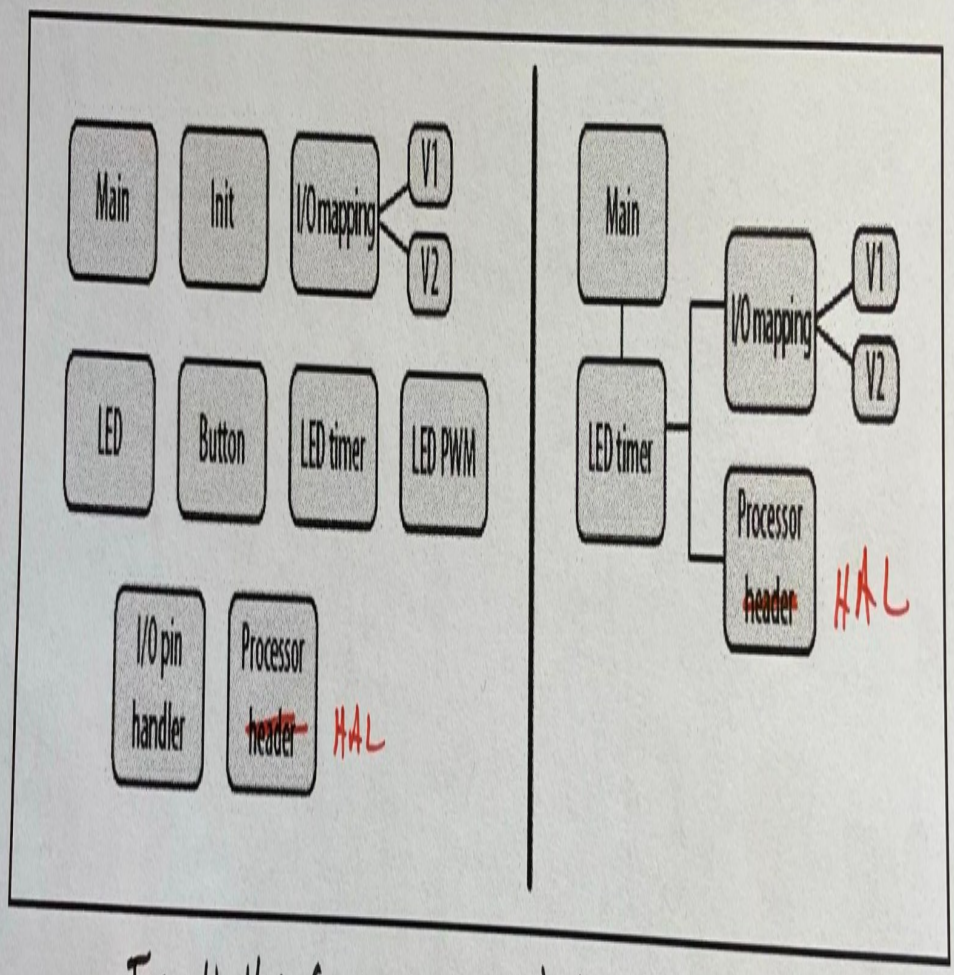


Fig 4-11: Comparing spaghetti prototyping code with a simple design

NOTE TO ARTIST: replace "processor header" with "processor HAL"

Figure 4-11. Comparing spaghetti prototyping code with a simpler design

In this example, many things are easy to remove because they just aren't needed. One thing that is harder to decide about is the dependency injection. It increases flexibility for future changes, which is a good reason for leaving it in. However, when you have to allocate a specific timer to a specific I/O pin, the configuration of the system becomes more processor-dependent and rigid. The cost of forcing it to be flexible can be high if you try to build a file to handle every contingency. In this case, I considered the idea and weighed the cost of making a file I'd never want to show anyone with the benefit of reducing the chance of writing bugs in the I/O subsystem. I chose to make more readable files, even if it means a few initial bugs, but I respect either option.

On the right side of Figure 4-11, the code base is trimmed down, using only the modules it needs. It keeps the I/O mapping header file, even with definitions for the old board because the cost is low (it is in a separate file for easy maintenance and requires no additional processor cycles). Embedded systems engineers tend to end up with the oldest hardware (karmic payback for the time at the beginning of the project when we had the newest hardware). You may need the old board header file for future development. Pretty much everything else that isn't critical can go into version control and then be removed from the project.

It hurts to see effort thrown away. But you haven't! All of the other code was necessary to make the prototypes that were required to make the product. The code aided marketing, and you learned a lot while writing and testing it. The best part about developing the prototype code is that the final code can look clean *because* you explored the options.

You need to balance the flexibility of leaving all of the code in with the maintainability of a code base that is easy to understand. Now that you've written it once, trust that your future self can write it again (if you have to).

Further Reading

- "A Guide to Debouncing" on Jack Ganssle's ganssle.com offers mechanics, real-world experimentation, other methods for implementing your debouncing code, and some excellent

methods for doing it in the hardware and saving your processor cycles for something more interesting. His whole website is worth some of your time.

- STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx Refernce Manual (RM0008), Rev 21, February 2021
- MSP430F2xx, MSP430G2xx Family User's Guide (slau144k), August 2022
- Atmel user manual: 8-bit Microcontroller with 2/4/8K Bytes In-System Programmable Flash (ATtiny25/V, ATtiny45/V, ATtiny85/V), Rev. 2586Q–AVR–08/2013.
- Atmel application note: AVR 130: Setup and Use the AVR Timers
- If bitwise operations are new to you, there are several games that can give you a more intuitive feel for how these work. Turing Complete is my current favorite. Look in this book's github repo for other suggestions.

INTERVIEW QUESTION: WAITING FOR A REGISTER TO CHANGE

What is wrong with this piece of code?

```
void IOWaitForRegChange(unsigned int* reg, unsigned int bitmask){
     unsigned int orig = *reg & bitmask;
     while (orig == (*reg & bitmask)) { /* do nothing */ ; }
    }
```

"What's wrong with this code?" is a tough question because the goal is to figure out what the interviewer thinks is wrong with the code. For this code, I can imagine an interviewee wondering where the comment header is. And whether there really should exist a function that waits forever without any sort of timeout or error handling.

If an interviewee flails, pointing out noncritical things, I would tell him that the function never returns, even though the register changes, as observed on an oscilloscope. If he continues to flounder, I would tell him that the code compiles with optimizations on.

In the end, this is not a see-how-you-think question, but one with a single correct answer: the code is missing the `volatile` keyword. To succeed in an embedded systems interview, you have to know what that keyword does (it is similar in C, C++, and Java).

Chapter 5. Managing the Flow of Activity

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at s*evans@oreilly.com*.

In Chapter 2, we looked at different ways to break up a system into manageable chunks. That chapter described the what and why of design; this chapter covers the how. Once you've identified the pieces, getting them all to work together as a system can be daunting.

Scheduling and Operating System Basics

Structuring an embedded system without an operating system requires an understanding of some of the things that an operating system can do for you. I'm only going to give brief highlights; if any of this first section is brand new to you, you may want to review a book about operating systems (see "Further Reading").

Tasks

When you turn on your computer, if you are like me, you load up an email client, web browser, and compiler. Several other programs start automatically (such as my calendar). Each of these programs runs on the computer, seemingly in parallel, even if you have only one processor.

WARNING

> Three words that mean slightly different things, but that overlap extensively, are sometimes used interchangeably. While these definitions are the ones I was taught, some RTOSs may switch them around. A *task* is something the processor does. A *thread* is a task plus some overhead, such as memory. A *process* typically is a complete unit of execution with its own memory space, usually compiled separately from other processes. I'm focusing on tasks; threads and processes generally imply an operating system.

The operating system you are running has a *scheduler* that does the switching between active processes (or threads), allowing each to run in its proper turn. There are many ways to implement schedulers, far beyond the scope of this book (let alone this small section). The key point is that a scheduler knows about all of the tasks/threads/processes/things your system should do and chooses which one it does *right now*.

Without an operating system, you are going to need to manage the scheduling yourself. The simplest scheduler has only one task. As with the blinking LED project in Chapter 4, you can do the same thing every time (on, wait, off, wait, repeat). However, things become more complex as the system reacts to changes in its environment (e.g., the button presses that change the LED timing).

Communication Between Tasks

The button-press interrupt and the LED-blinking loop are two tasks in the tiny system we have been examining. When we used a global variable to indicate that the button state changed, the two tasks communicated with each other. However, sharing memory between tasks is dangerous; you have to be careful about how you do it.

Figure 5-1 shows the normal course of events where the interrupt sets the shared memory when a button is pressed. Then the main loop reads and clears the variable. The user presses the button again, which causes the interrupt to set the memory again.

Suppose that just as the main loop is clearing the variable, the interrupt fires again. At the very instant that the variable is being cleared, an interrupt stops the processing, swooping in to set the variable. Does it end up as set or cleared?

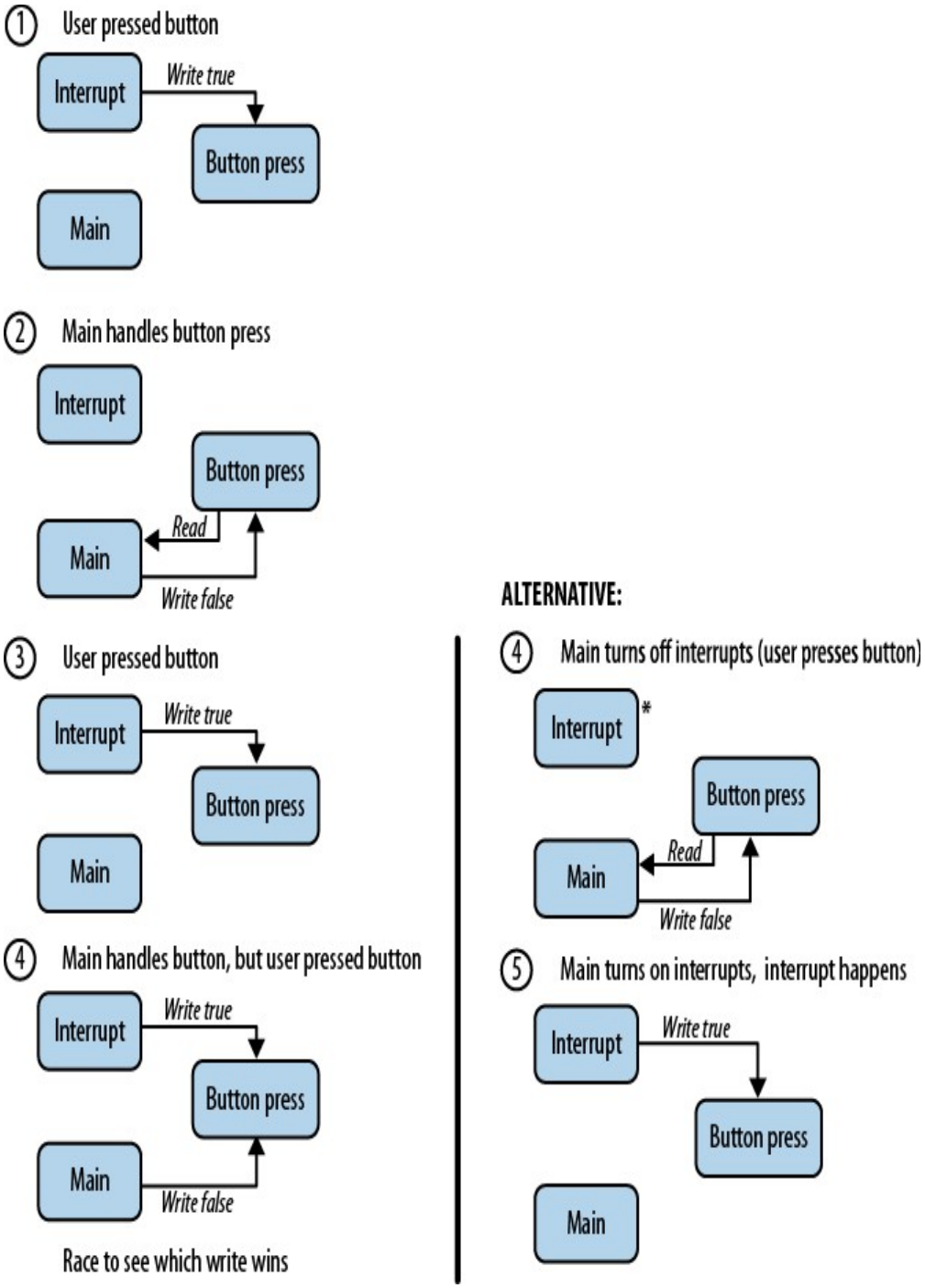


Figure 5-1. Race condition in shared memory

The answer depends on the precise timing of what happens. This uncertainty is a big problem. If it can be this complicated with a simple Boolean value, consider what could happen if the code needs to set two variables or a whole array of data.

When this happens, it is called a *race condition*. Any memory shared between tasks can exhibit this uncertainty, leading to unstable and inconsistent behavior.

In this example, given the way the interrupt and button work together, it is likely that the system will miss a button press (if the interrupt wins the race to set the variable, the main function will clear it, even though it should be set). Though only a slight annoyance to a user in this particular system, race conditions can lead to unsafe conditions in a more critical system.

Avoiding Race Conditions

We need a way to prohibit multiple tasks from writing to the same memory. It isn't only writing that can be an issue; the main loop reads both the variable that says the button is changed and the value of the button. If an interrupt occurs between those two reads, it may change the value of the button between them.

Any time memory shared between tasks is read or written, it creates a *critical section* of code, meaning code that accesses a shared resource (memory or device). The shared resource must be protected so only one task can modify it at a time. This is called *mutual exclusion*, often shortened to *mutex*. They are sometimes implemented with *semaphores*, *mailboxes*, or *message queues.*

In a system with an OS, when two tasks are running but neither is an interrupt, you can use a mutex to indicate which task owns a resource. This can be as simple as a variable indicating whether the resource (or global variable) is available for use. However, when one of the two tasks is an interrupt, we have already seen that not even a Boolean value is safe, so this resource ownership change has to be *atomic*. An atomic action is one that cannot be interrupted by anything else in the system.

From here on, we are going to focus on systems where a task is interruptible but otherwise runs until it gives up control. In that case, race conditions are avoided by disallowing interrupts while accessing the shared global variables. This has a downside, though: when you turn off interrupts, the system can't respond as quickly to button presses, because it has to wait to get out of a critical section. Turning off interrupts increases the *system latency* (time it takes to respond).

Latency is important as we talk about *real-time systems*. A real-time system must respond to an event within a fixed amount of time. Although the required response time depends on the system, usually it is measured in microseconds or milliseconds. As latency increases, the time it takes before an event can be noticed by the system increases, and so the total time between an event and its response increases.

Priority Inversion

Some processors allow interrupts to have different priorities (like operating systems do for processes and threads). Although this flexibility can be very useful, it is possible to create a lot of trouble for yourself. Figure 5-2 shows a typical operating system's definition of priority inversion. It is OK for a high-priority process to stop because it needs access to something a low-priority process has. However, if a medium-priority process starts, it can block the low-priority process from completing its use of the resource needed by the high-priority task. The medium-priority task blocks the high-priority task.

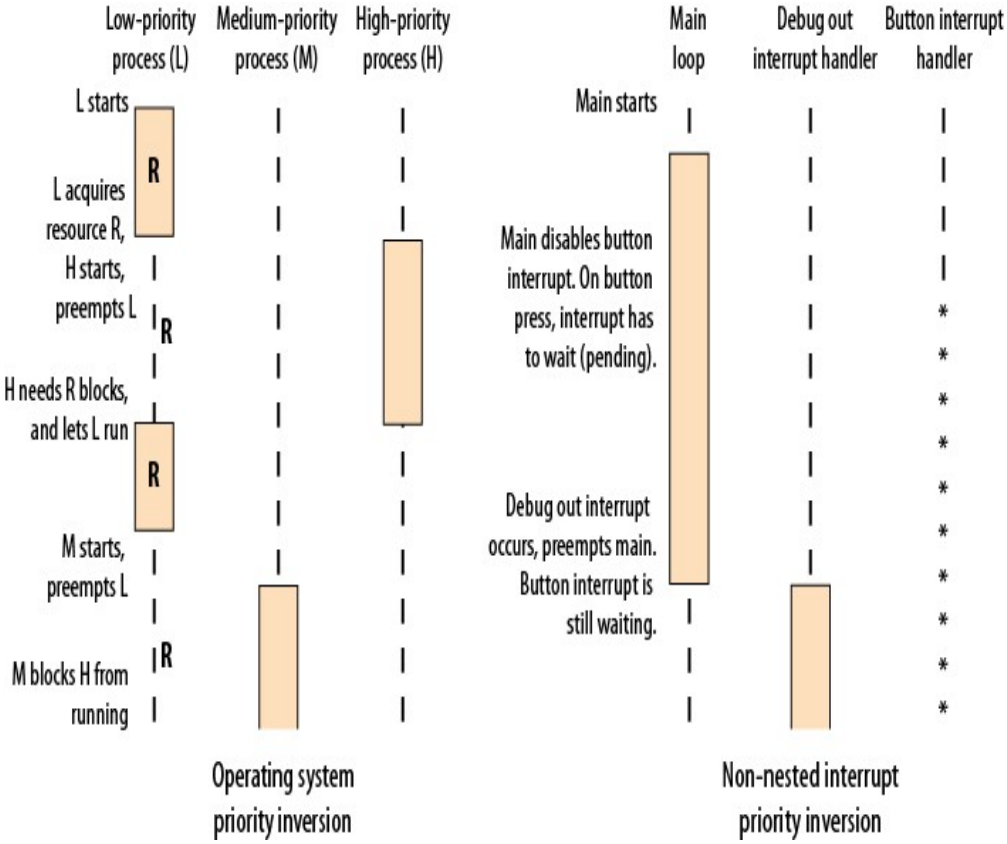


Figure 5-2. Priority inversion with an operating system and with interrupts

For example, say we have the high-priority button-press interrupt and our low-priority main loop. When the main function accesses the button-press variable, it turns off the button-press interrupt to avoid a race condition. Later, we add another interrupt to output debug data through a communication port. This should be a background task of medium priority, something that should get done but shouldn't block the system from handling a button press. However, if the debug task interrupts the main loop, it is doing exactly that.

What is the most important thing for the processor to be doing? That should be the highest priority. If someone had asked whether debug output is more important than button presses, we would have said no. If that is true, why is the processor running the debug interrupt and not the button handler? The simple fix in this case is to disable *all* interrupts (or all interrupts that are lower in priority than the button handler) when the button interrupt occurs.

As we look at different ways of task management without an operating system, be alert for situations that cause the processor to inadvertently run a task that isn't the highest priority.

State Machines

One way to keep your system organized while you have more than one thing going on is to use a *state machine*. This is a standard software pattern, one that embedded systems use a lot. According to *Design Patterns: Elements of Reusable Object-Oriented Software*, the intent of the State pattern is to "allow an object to alter its behavior when its internal state changes. The object will appear to change its class."

Put more simply, when you call a state machine, it will do whatever it thinks it should do based on its current state. It will not do the same thing each time, but will base the change of behavior on its *context*, which consists of the environment and the state machine's history (internal state). If this all sounds clinical and theoretical, there is an easier way to think about state machines: flow charts.

Almost any state machine can be represented as a flow chart. (Conversely, a problem you solve with a flow chart is probably destined to be a state machine.) State machines can also be represented as finite state automata (as

shown in Figure 5-3), where each state is a circle and a change between states (a state transition) is an arrow.

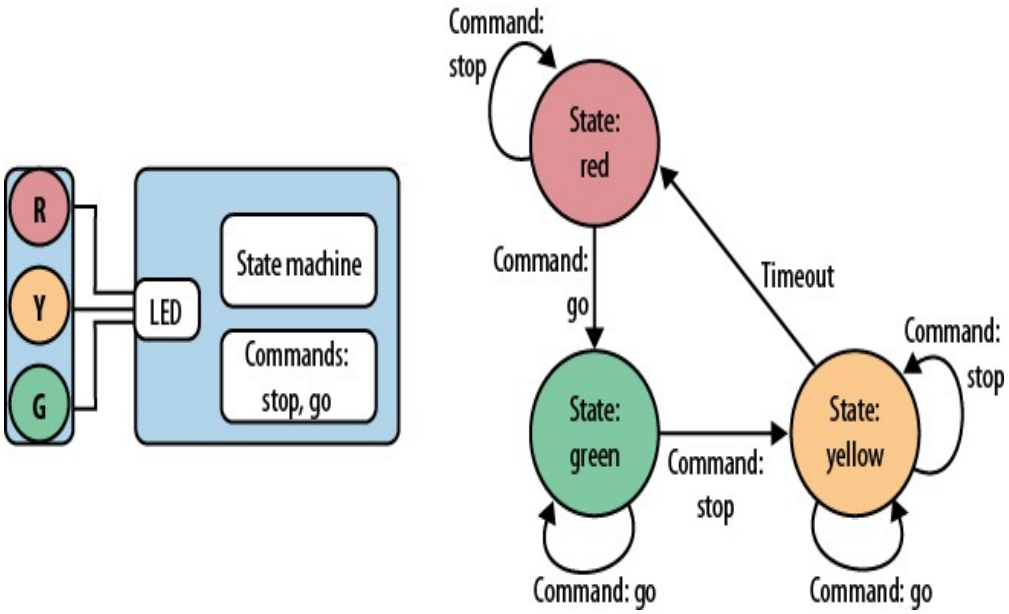


Figure 5-3. Stoplight system

We'll look at each element of this figure in the following section. The arrows in the diagram are as important as the circles. State machines are not only about the states that a system can occupy, but also about the events that the system handles.

State Machine Example: Stoplight Controller

To talk about state machines properly, we need an example that is a bit more complicated than a blinking LED and a button. Figure 5-3 shows a stoplight controller. When the light is red and the controller gets a message to `go`, it turns the light green. When the controller gets a message to `stop`, it turns the light yellow for a time and then red. We've implemented this stoplight so it stays green as long as no one needs it to change. Presumably, a command to stop will be generated when a car arrives on the cross street.

One state transition—which is the formal term for what each arrow shows—is a bit subtle. When the light is yellow and the controller receives a message to `go`, it should not change the light back to green. Yellow lights should change to red, not green, so the message to `go` just leaves the light in the yellow stage. This subtlety is an example of the gotchas that state machines create for you. You have to be prepared for every event that can happen in

every state, including very rare cases—even cases that shouldn’t take place but that may happen because of errors.

To work with a state machine, the first thing to do is to figure out the states and the events that can change states. With a system as simple as this, drawing a diagram is the best thing to do. Once you’ve identified the states (red, yellow, green), look at its connections. The `stop` and `go` commands should happen only in the green and red states (respectively). Even so, the commands are asynchronous, coming from outside the system, so each state should be able to handle them, even if they occur improperly. The easiest way of handling improper commands here is to ignore them, putting a loop back to the associated state.

WARNING

If an improper event occurs, it may be better to generate an error of some sort. Use your judgment.

State-Centric State Machine

Most people think of a state machine as a big if-else statement, or switch statement:

```
while (1) {
  look for event

  switch (state) {
  case (green light):
    if (event is stop command)
      turn off green light
      turn on yellow light
      set state to yellow light
      start timer
    break;
  case (yellow light):
    if (event is timer expired)
      turn off yellow light
      turn on red light
      set state to red light
    break;
  case (red light):
    if (event is go command)
      turn off red light
      turn on green light
      set state to green light
    break;
  default (unhandled state)
    error!
```

```
    }
  }
```

The form of the state machine here is:

```
    case (current state):
      if event valid for this state
        handle event
        prepare for new state
        set new state
```

The state machine can change its context (move to a new state). This means that each state needs to know about its sibling states.

State-Centric State Machine with Hidden Transitions

Another way to implement the state machine is to separate the state transition information from the state machine. This is theoretically better than the model in the previous section because it has more encapsulation and fewer dependencies. The previous model forced each state to know how and when to trigger each of the other states that can be reached from it. In the model I'll show in this section, some higher-level system keeps track of which state reaches which other state. I'll call this the "next state" function. It handles every state and puts the system into the next state that it should be in. By creating this function, you can separate the actions taken in each state from the state transitions.

The generic form of this would look like this:

```
    case (state):
      make sure current state is actively doing what it needs
      if event valid for this state
        call next state function
```

In the stoplight example, this model would create simpler code for each state, and it would also be similar for almost all states. For instance, the green light state would look like this:

```
    case (green light):
      turn on green light (even if it is already on)
      if (event is stop)
        turn off green light
        call next state function
      break;
```

The "`next state`" function should be called when a change occurs. The code will be familiar from the previous section because it is a simple switch statement as well:

```
next state function:
    switch (state) {
    case (green light):
```

```
      set state to yellow light
      break;
    case (yellow light):
      set state to red light
      break;
    case (red light):
      set state to green light
      break;
```

Now you have one place to look for the state and one for the transitions, but each one is pretty simple. In this example, the state transitions are independent of the event, one clue that this is a good implementation method.

This model is not always the best. For example, if a go command in the yellow state led back to green, the next state function would depend on both the current state and event. The goal of this method is to hide transitions, not to obfuscate your code. Sometimes it is better to leave it all together (as described in the previous section).

Event-Centric State Machine

Another way to implement a state machine is to turn it on its side by having the events control the flow. With this approach, each event has an associated set of conditionals:

```
    case (event):
      if state transition for this event
        go to new state
```

For example:

```
    switch (event)
    case (stop):
      if (state is green light)
        turn off green light
        go to next state
      // else do nothing
      break;
```

Most state machines create a comfortable fit between switch statements and states, but in some cases it may be cleaner to associate the state machines with events, as shown here. The functions still might need a switch statement to handle the dependency on the current state:

```
function to handle stop event
  if (state == green light)
    turn off green light
    go to next state
```

Unlike the state-centric options, the event-centric state machine implementation can make it difficult to do housekeeping activities (such as checking for a timeout in the yellow state). You could make housekeeping an

event if your system needs regular maintenance, or you could stick with the state-centric implementation shown in the previous section.

State Pattern

An object-oriented way to implement a state machine is to treat each state as an object and create methods in the object to handle each event. Each state object in our example would have these member functions:

`Enter`

: Called when the state is entered (turns on its light)

`Exit`

: Called when leaving the state (turns off its light)

`EventGo`

: Handles the go event as appropriate for the state

`EventStop`

: Handles the stop event

`Housekeeping`

: Periodic call to let the state check for changes (such as timeouts)

A higher-level object, the `context`, keeps track of the states and calls the appropriate function. It must provide a way for the states to indicate state transitions. As before, the states might know about each other and choose appropriately, or the transitions may be hidden, so the state might only indicate a need to go to the next state. Our system is straightforward, so a simple next-state function will be enough. In pseudocode, the class looks like:

```
class Context {
  class State Red, Yellow, Green;
  class State Current;

constructor:
  Current = Red;
  Current.Enter();

destructor:
  Current.Exit();

Go:
  if (Current.Go() indicates a state change)
    NextState();
```

```
Stop:
  if (Current.Stop() indicates a state change)
    NextState();

Housekeeping:
  if (Current.Housekeeping() indicates a state change)
    NextState();

NextState:
  Current.Exit();
  if (Current is Red)    Current = Green;
  if (Current is Yellow) Current = Red;
  if (Current is Green)  Current = Yellow;
  Current.Enter();
}
```

Allowing each state to be treated exactly the same frees the system from the switch statement, letting the objects do the work the conditionals used to do.

Table-Driven State Machine

Although flow charts and state diagrams are handy for conceiving of a state machine, an easier way to document and fully define a state machine is to use a table.

In the example table in Figure 5-4, each state has a row with an action and multiple events. The action column describes what should occur in that state (which particular light should be on). The other columns show the transition that needs to occur when the system is in a state and an event occurs. Often the transition is simply to move to a new state (and perform the new associated action).

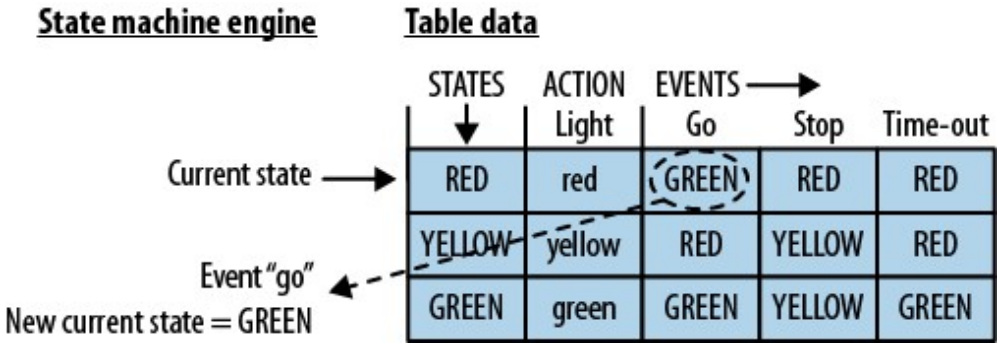


Figure 5-4. The state machine as a data table

NOTE

When I worked on children's toys, they often offered 30 or more buttons (ABCs, volume, control, etc.). A table like this helped us figure out which events didn't get handled in the flow chart. Even though a button may be invalid for a state, someone somewhere will still

inexplicably press it. So tables like these not only aided implementation and documentation, they were critical to designing the game play.

Defining the system as a table here hints toward defining it as a data table in the code. Instead of one large, complex piece of code, you end up with smaller, simpler pieces:

- a data table showing the action that occurs in a state and what state to go to when an event happens
- an engine that reads the data table and does what it says

The best part is that the engine is reusable, so if you are going to implement many complex state machines, this is a great solution.

The stoplight problem is a little too simple to make this method worthwhile for implementation, but it is a straightforward example. Let's start with the information in each table:

```
struct sStateTableEntry {
  tLight light;       // all states have associated lights
  tState goEvent;     // state to enter when go event occurs
  tState stopEvent;   // ... when stop event occurs
  tState timeoutEvent;// ... when timeout occurs
};
```

In addition to the next event for each table, I put in the light associated with the current state so that each state can be handled exactly the same way. (This state machine method really shines when there are lots of states that are all very similar.) Putting the light in the state table means our event handlers do the same thing for each state:

```
// event handler
void HandleEventGo(struct sStateTableEntry *currentState){
  // turn off the light (unless we're just going to turn it back on)
  if (currentState->light != currentState->go.light) {
    LightOff(currentState->light);
  }
  currentState = currentState->go;
  LightOn(currentState->light);
  StartTimer();
}
```

What about the actual table? It needs to start by defining an order in the data table:

```
typedef enum { kRedState = 0, kYellowState = 1, kGreenState = 2 } tState;

struct sStateTableEntry stateTable[] = {
```

```
  { kRedLight,    kGreenState,  kRedState,    kRedState},   // Red
  { kYellowLight, kYellowState, kYellowState, kRedState},   // Yellow
  { kGreenLight,  kGreenState,  kYellowState, kGreenState}, // Green
}
```

Typing in this table is a pain, and it is easy to get into trouble with an off-by-one error. However, if your state machine is a spreadsheet table and you can save it as a comma- or tab-separated variable file, a small script can generate the table for you. It feels a bit like cheating to reorganize your state machine as a spreadsheet, but this is a powerful technique when the number of states and events is large.

Making the state machine a table creates a dense view of the information, letting you take in the whole state machine at a glance. This will not only help you see holes (unhandled state/event combinations), it will also let you see that the product handles events consistently.

The table can show more complex interactions as well, detailing what needs to happen. However, as complexity increases, the table loses the ability to simply become data, and it likely needs to return to one of the implementation mechanisms shown earlier, one oriented around control flow statements. For example, if we add some detail and error-finding to the stoplight, we get a more complex picture:

State/events	Command: `go`	Command: `stop`	Timeout
Red	Move to green	Do nothing	Invalid (log error), do nothing
Yellow	Do not clear event (defer handling to red)	Do nothing	Move to red
Green	Do nothing	Move to yellow, set timer	Invalid (log error), do nothing

Even when the state machine can’t be implemented as a data-table-driven system, representing it as a table provides better documentation than a flow chart.

Choosing a State Machine Implementation

Each option I showed for a state machine offers the same functionality, even though the implementations are different. When you consider your implementation, be lazy. Choose the option that leads to the least amount of code. If they are all about the same, choose the one with the least amount of *replicated* code. If one implementation lets you reuse a section of code without copying it, that's a better implementation for your system. If that still doesn't help you choose, consider which form of code will be the most easily read by someone else.

State machines are powerful because they let your code act according to its history (state) and the environment (event). However, because they react differently depending on those things, they can be very difficult to maintain, leading to spaghetti code and dependencies between states that are not obvious to the casual observer. On the other hand, there is no better way to implement a reactive system, no way that makes the inherent complexity easier to maintain. Documentation is key, which is why I focused in these sections on the human-readable representation of the state machine before showing a code implementation.

Interrupts

In our system so far, we haven't worried about how the events occur. The state machine doesn't care whether there is a person pushing buttons that say "stop" and "go" or there is a wireless Ethernet controller parsing a data stream looking for these commands. This sort of encapsulation is great for the state machine. Now, though, it is time to consider the events.

Interrupts can be scary. They are one of the things that make embedded systems different from traditional application software. Interrupts swoop in from nowhere to change the flow of the code. They can safely call only certain functions (and usually not the debug functions). Interrupts need to be fast, so fast that they are a piece of code that is still occasionally written in assembly language. And bugs in interrupts are often quite difficult to find because, by definition, they occur *asynchronously* (outside the normal flow of execution).

However, interrupts are not the bogeyman they've been made out to be. If you understand what happens when an interrupt occurs, you'll find where they can be a useful part of your software design.

I want you to remember that processors and interfaces are like software APIs (see “Reading a Datasheet” in Chapter 3) and that function pointers aren’t scary (see “Function Pointers Aren’t So Scary”also in Chapter 3). You’ll need both of those ideas in your head as we walk through what happens when an interrupt occurs:

1. An interrupt request (*IRQ*) happens inside the processor, triggered by a peripheral, the software, or a fault in the system.
2. The processor saves the *context* which includes where it was and the local variables.
3. The processor looks in the interrupt *vector table* to find the function associated with the interrupt.
4. The callback function runs (aka *interrupt service routine* or *ISR* or interrupt handler).
5. The processor restores the saved context, returning to the point before the interrupt occurred, and the program continues to run.

Let’s go through the process of interrupt handling in more detail.

An IRQ Happens

Some interrupts are like small high-priority tasks. Once their ISR is complete, the processor restores the context it saved and continues on its way as though nothing had happened. Most peripheral interrupts are like that: input lines, communication pathway, timers, peripherals, ADCs, etc.

Other interrupts are more like exceptions, handling system faults and never returning to normal execution. For example, an interrupt can occur when there is a memory error, there is a divide by zero error, the processor tries to execute an invalid instruction, or the power level is not quite sufficient to run the processor properly (brownout detection). Since these errors mean the program is invalid and the processor can’t run properly, often they are handled with infinite loops or processor resets.

Most interrupts you need to handle will be more like tasks than exceptions. They will let your processor run multiple tasks, seemingly in parallel.

Usually an interrupt request happens because you've configured the interrupt. If it wasn't you, then your vendor's startup code probably did the work. While often invisible to high-level language programmers, the startup code configures some hardware-oriented things (such as setting up the default interrupts, usually the fault interrupts, and configuring system clocks).

For task-like interrupts, your initialization code can configure them to occur under the conditions you specify. What exactly those conditions are depend on your processor. Your processor's user manual will be critical to setting up interrupts. (More on configuration in "Configuring Interrupts" later in this chapter, after we go through the parts of the interrupt itself.)

In our stoplight example, we want a timer interrupt to signal an event when the yellow light has been on for long enough. Because processors are different, I'll focus on the STM32F103xx processor, which is middle of the road in its interrupt-handling complexity (Microchip's ATtiny AVR is much simpler, and the TI MSP430x2xx is more complex).

Nonmaskable interrupts

Some processors define certain interrupts as so important that they can't be disabled; these are called *non-maskable interrupts (NMI)*. (I suppose not-disable-able interrupts was a little unwieldy to say.)

The processor faults noted earlier are one form of NMI. Often there is one I/O pin that can be linked to an NMI, which usually leads to an "on" button on the device. These interrupts cannot be ignored at any time and must be handled immediately, even in critical sections of code.

Interrupt priority

As noted in the section "Scheduling and Operating System Basics", some processors have a priority system for interrupts. For example, in our stoplight controller, a timer interrupt could be used to indicate when the yellow state should be completed, turning the light to red. This timer interrupt is pretty important. The other lights in the intersection have to stay in sync. If one of the controllers remained yellow when the cross traffic turned to green, it would cause accidents.

Some processors handle interrupt priority by virtue of the peripherals themselves (so timer 1 has a higher priority than timer 2, which in turn is more important than timer 3). Other processors allow you to configure the priority on a per-interrupt basis.

Nested interrupts

Some processors allow interrupts to be interrupted by another interrupt. Instead of priority being important only when an ISR is called, the priority is used to determine whether a new interrupt can preempt the one currently running.

This is a powerful tool that is likely to cause unnecessary complexity. Unless nested interrupts solve a clear problem particular to your system, it is customary to disable other interrupts while in an interrupt service routine. Most processors that allow nested interrupts give you a method for disabling them at system initialization.

STACKS (AND HEAPS)

A stack is a data structure that holds information in a last-in-first-out (LIFO) manner, as shown in Figure 5-5. You push data onto the stack (adding it to the stack's memory and increasing the pointer to where the next set of data will go). To get the last piece of data out, you pop the stack (which decreases the pointer).

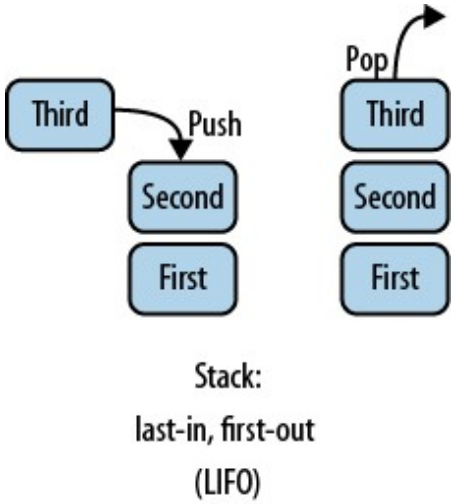


Figure 5-5. Stack basics

A stack is a simple data structure any developer can implement, but *the stack* (note the definite article) refers to the call stack that lies behind

every running program in a designated section of RAM. For each function called, the compiler creates a stack frame that contains the local variables, parameters, and the address to return to when the function is finished. There is a stack frame for every function call, starting with the reset vector, then the call to main, and then whatever you call after that. (Or if you have multiple threads, each one will have its own stack.)

A heap is a tree data structure. *The heap* is where dynamically allocated memory comes from (so named because it can be implemented as a heap data structure).

Usually the heap grows up (see Figure 5-6), whereas the stack grows down. When they meet, your system crashes. Well, actually your system will probably crash before they meet because there may be other things between them (global and static variables and possibly code). (More about this in Chapter 8.)

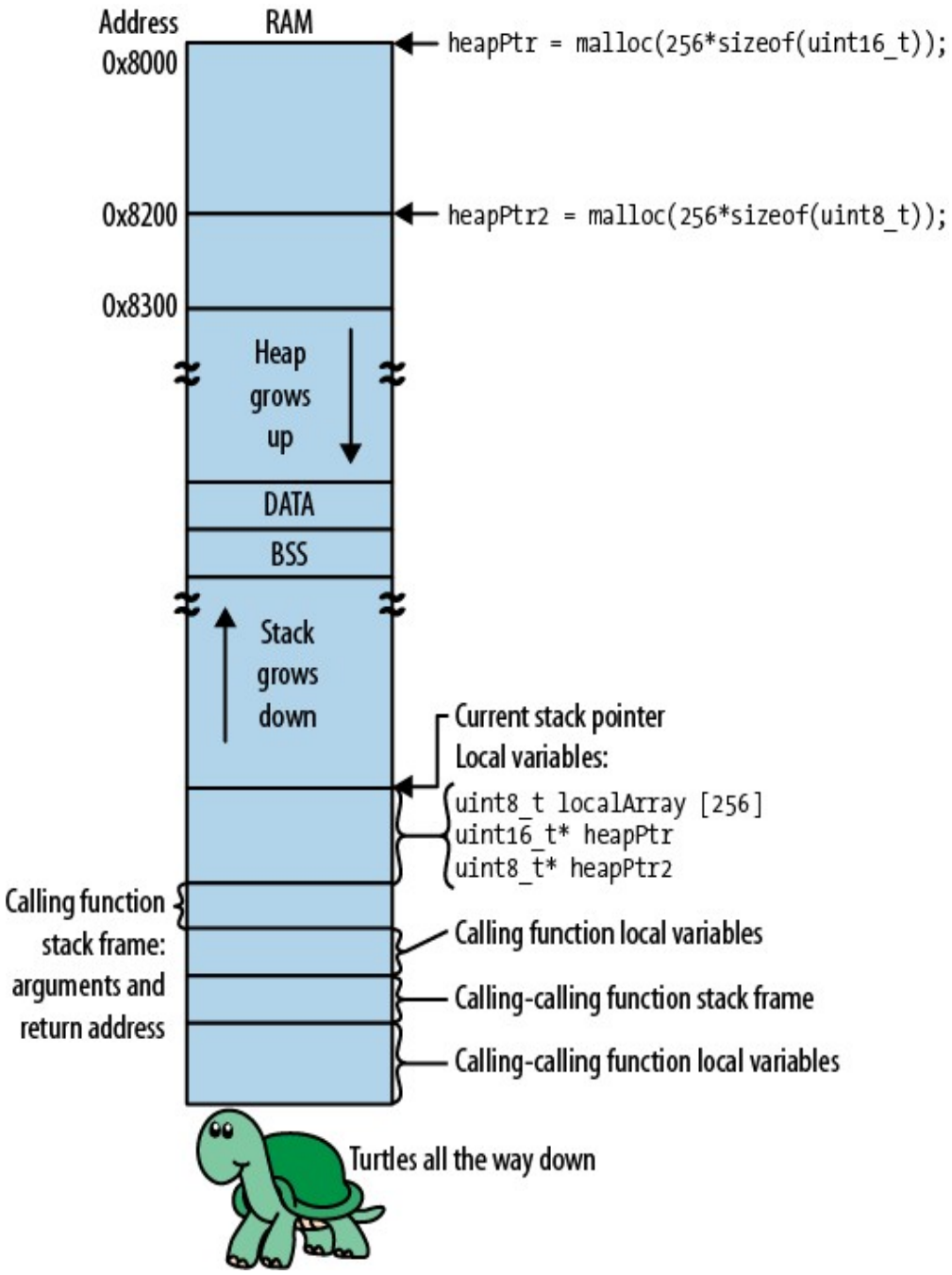


Figure 5-6. Memory map of a system

If the stack gets too large, it can grow into other areas of memory, for example, where a global array is located. If the stack overwrites the memory of the array, the corrupt data in the array is likely to give you incorrect results. This is called a *stack overflow*. On the other hand, if the global array overwrites the stack, the function return address may be corrupted. When the function returns, the program will return to a bogus address and crash (usually due to an inappropriate instruction).

Save the Context

After the interrupt request happens (after you've set it up and the event happens), the processor saves where it was before it finds the appropriate ISR and calls it. Like a bookmark saving your spot, the processor saves its context to the stack; see "Stacks (and Heaps)". The context includes the program counter (which points to the next instruction to execute) and a subset of the processor registers (which are like the processor's own cached local RAM). The register contents are saved so that the interrupt code can use them in its execution without corrupting the state of the primary code path.

These steps don't come for free. The amount of time it takes between the IRQ and the ISR is the processor's *interrupt latency*, usually advertised in the user manual as taking a certain number of processor cycles.

The *system latency* is the worst-case amount of time it takes from when an interrupt event happens to the start of the ISR. It includes the processor's interrupt latency and the maximum amount of time interrupts are ever disabled.

NOTE

If interrupt nesting is disallowed, then new interrupts are disabled while an interrupt is being handled. The largest contributor to system latency may be the time spent in the longest interrupt. Keep interrupts short!

The interrupt latency processor cycles are lost. If you had a processor that could do a hundred instructions a second (100 Hz) with an interrupt latency of 10 cycles and you set up a timer that interrupted every second, you'd lose 10% of your cycles to context switching. Actually, since you have to restore the context, it could be worse than that. Even though 10 cycles is a decent interrupt latency (not great, but not bad), 100 Hz systems are rare. However, doing this math with a 30 MHz processor, handling audio in an interrupt at 44,100 Hz with a 10-cycle latency uses 1.47% of the processor's cycles simply calling the interrupt handler. That doesn't even include the time spent executing the ISR or returning from it.

Processor designers work to keep the interrupt latency low. One way to do that is to store the minimum amount of context. This means that instead of

saving all of the processor registers, the processor will save only a subset, requiring the software to store the other ones it needs. This is usually handled by the compiler, increasing the effective system latency.

Calculating system latency

Calling lots of functions while in an interrupt is discouraged. Each function call has some overhead (discussed more fully in Chapter 8), so function calls make your interrupt take longer. If you have other interrupts disabled during your interrupt, your system latency increases with each function call.

As your system latency increases, its ability to handle events in real time also decreases. Going back to the 30 MHz processor with its 44,100 Hz interrupt, if each interrupt uses 10 cycles for interrupt overhead and 10 cycles calling five short functions (each) that collectively use 275 cycles actually processing information, no other interrupt can be handled for at least 335 cycles (10 + 5*10 + 275). The time consumed by this is 11us (335/30 MHz). Also note that the system spends nearly 50% of its time in the interrupt (44,100*11 us=0.492 s). Reducing the overhead of the interrupt will free up processor cycles for other tasks.

Get the ISR from the Vector Table

After the IRQ happens and the context is saved, the third step for handling an interrupt is to determine which ISR to call by looking in the *interrupt vector table (IVT)*. This table is located at a specific area of memory on your processor. When an interrupt occurs, the processor looks in the table to call the function associated with the interrupt.

The interrupt vector table is a list of callback functions, one for each type of interrupt. Secretly, the vector table is a list of function pointers, though it is often written in assembly.

Initializing the Vector Table

The start-up code (which probably came with your compiler or processor vendor HAL) sets up the vector table for you, usually with dummy interrupt handlers. When debugging, you probably want your unhandled interrupts to loop so that you find them and turn off the interrupt or handle it properly.

When the product hits the market, having an unhandled interrupt handler go into an endless loop will cause your system to become unresponsive. You may want unhandled interrupts to simply return to normal execution. It wastes processor cycles and is still a bug, but at least the bug's effect on the customer is a slight slowdown instead of a system that needs a reboot.

In some cases, if you name your handler function the same as the one in the table, some tools magic will make the linker use yours in the IVT. With other processors and compilers, though, you will need to insert your ISR into the function table at the correct slot for your interrupt.

In the STM32F103xx processor, using the STM32Cube HAL, we'd only need to name the ISR `TIM2_IRQHandler` and the linker would put the correct function address in the vector table. In some processors, such as Microchip's AT91SAM7S, you have to create a function and put it in the table manually:

```
interrupt void Timer1ISR(){
  …
}

void YellowTimerInit(){
  …
  AT91C_BASE_AIC->AIC_SVR[AT91C_ID_TC1] = // Set TC1 IRQ handler address in the IVT
      (unsigned long)Timer1ISR;     // Source Vector Register, slot AT91C_ID_TC1 (12)
  …
}
```

This has to occur when the interrupt is initialized and before the interrupt is enabled.

Looking up the ISR

The vector table is located at a particular address in memory, set by the linker script. Since this is probably done for you in the start-up code, usually you don't need to know how to do it. However, in case you are curious, let's dig into the start-up code for the STM32F103xx for a moment:

```
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) = {
  // Core Level - CM3
  &_estack,           // The initial stack pointer
  Reset_Handler,      // This is the very beginning of running
  NMI_Handler,         // Non-maskable fault handler


…
  TIM2_IRQHandler,  // 28, TIM2 global interrupt
  TIM3_IRQHandler,  // 29,
  TIM4_IRQHandler,  // 30,
…
}
```

First in the listing is a nonstandard line that communicates to the compiler that the following variable needs to go someplace special and that the linker script will indicate where it is (at the location specified by `.isr_vector`). We'll talk more about linker scripts later (see "Linker Scripts" in Chapter 8), but right now it is safe to say that the `.isr_vector` linker variable is located where the user manual says the vector table should be (0x00000000).

Inside the array of `void *` elements, there is the location for the stack, also set in the linker script. After that is the reset vector, which is the address of the code that is called when your system boots up or resets for another reason. It is one of the exception interrupts mentioned earlier. Most people don't think of turning on the power or pushing the reset button as an interrupt to their code. However, the processor responds to a reset by loading a vector from the table, in the same way that it responds to an interrupt by loading a vector. Later in the table, there are the peripheral interrupts, such as our timer. Figure 5-7 shows how this would look in the processor's memory.

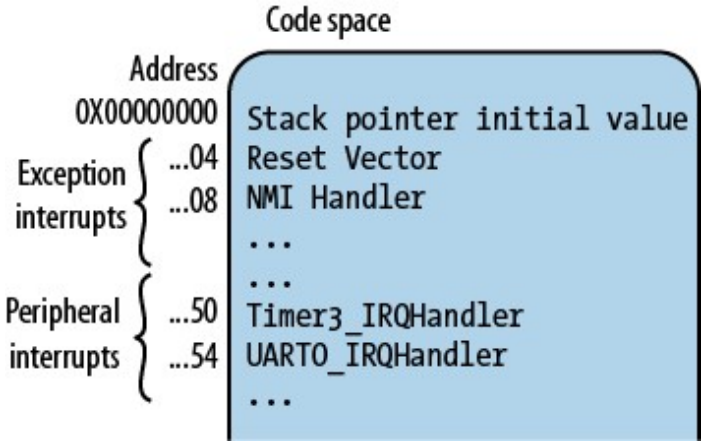


Figure 5-7. Vector table in memory

Each interrupt has an interrupt number. For our timer 3 interrupt, it is 50. When the interrupt happens, it is signaled by just this number, and the part of the processor that generates the IRQ sends it to the part of the processor that looks up the handler in the vector table as the number. The processor uses the interrupt number and the structure of the vector table to look up the address to call when the interrupt arrives.

Calling the ISR

So far we've gone through what you need to know about the workings of an ISR and how to set it up. Now we'll get to the meaty part of actual ISR that

you'll need to implement. We've already seen the most important rules surrounding ISRs:

- Keep ISRs short because longer ones increase your system latency. Generally avoid function calls, which may have hidden depths and increase overhead.
- Don't call non-reentrant functions (such as `printf`), because global variables can be corrupted by interrupts.
- Turn off other interrupts during the ISR to avoid priority inversion problems.

This gives us the design guidelines to implement an ISR. Our timer interrupt is easy to implement because all we need to do is set a flag indicating it is time to change to a red light:

```
volatile tBoolean gYellowTimeout = FALSE; // global variable set by the interrupt
                                          // handler, which is cleared by normal code
                                          // when event handled
void TIM2_IRQHandler(void){
  __disable_irq();        // disallow nesting of interrupts
  gYellowTimeout = TRUE;
  __enable_irq();
}
```

The interrupt doesn't print out a debug message or change the state to red. As much as it would make sense to do that, keeping it short means we need to let other parts of the system take care of these things.

Our system so far has been entirely devoted to controlling the lights at the signal, so we haven't had to worry about system latency. However, few controllers in the wild have such a limited scope, so for this section we'll add some functionality. Say the stoplight controller is also taking pictures of red-light runners with a traffic camera, dispensing gum balls to good citizens who use the crosswalk, and studying to pass the Turing test. With the small processor so overloaded, we might have to make allowances for the state machine not running as quickly as we'd like.

After reviewing the product requirements and making sure we can't alleviate the problem by reducing the feature set (do we really need sentient stoplights?), we might opt to make sure the system is left in the safest possible mode by turning off the yellow light and turning on the red light in the ISR.

Since the state machine is non-reentrant, the interrupt would have to circumvent it and force the lights to the new color, getting them out of sync with the state recorded by the state machine. Not only does this simple timeout ISR become coupled to the event handling and the light code, it creates an oddity in the system that is a little hard to explain to someone taking over maintenance. That is why I'd verify the requirements before implementing the workaround. On the other hand, interrupts can do more than set flags and unblock communications drivers. Using them to make a system safer is a worthy trade-off (although fixing the design to avoid the heavyweight state change would still be preferable).

Many processors require that you acknowledge (or clear) the interrupt. Usually this is accomplished by reading a status register. As noted in "Writing an Indirect Register", this may have the side effect of clearing the bit.

WRITING AN INDIRECT REGISTER

Memory-mapped registers can have some interesting properties. Registers don't always act like variables.

If you want to modify a normal variable, the steps hidden in your `memoryReg |= 0x01` line of code are:

1. Read the current value of the memory into a processor register.
2. Modify the value.
3. Write the variable back into memory.

However, for a register that sets interrupt handling, this multistep process can cause problems if an interrupt occurs between any two of these steps. To prevent this race condition—where an interrupt occurs during an inconsistent state—actions on registers must be atomic (executing in a single instruction).

This is accomplished by having a set of *indirect* registers: a register that can set bits and a register that can clear bits, each of which acts on a third functional register that may not be directly writable (nor memory mapped). To indirectly set a bit, you have to write the bit in the set

register. To indirectly clear a bit, you write that bit in the clear register. In either register (set or clear), the unset bits don't do anything to the functional register. (Part B of Figure 5-8 gives an example of setting and clearing bits and the contents of the functional register at each step.)

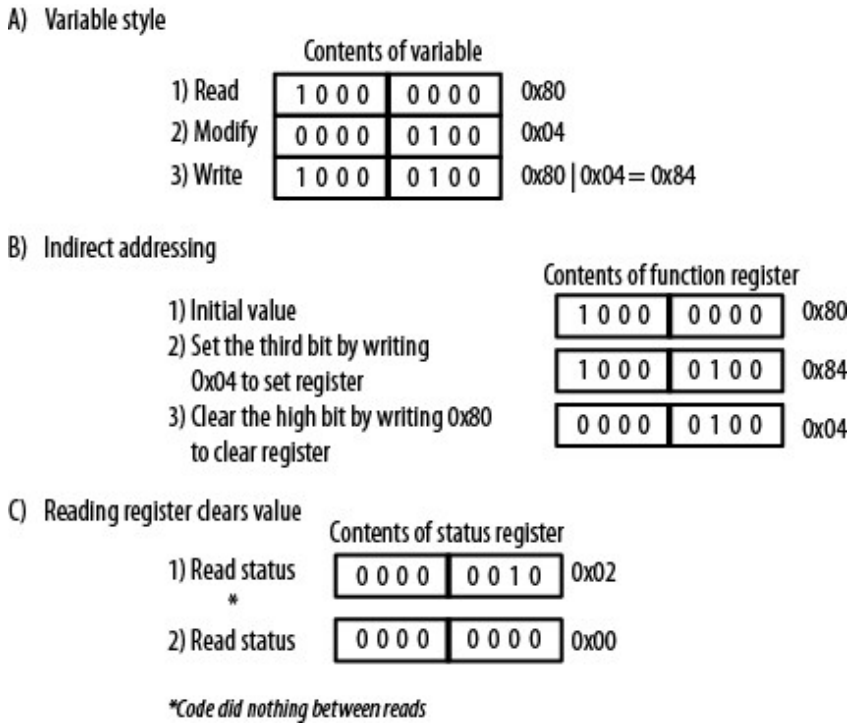


Figure 5-8. Methods for setting register values

Sometimes the functional register is reflected in the set and clear registers, so if you read either one of those, you can see the currently set bits. It can be confusing, though, to read something that is not what you have written to the same register.

A functional register also can be write-only without any way to be read. Some processors save memory space by having a register act as a functional register when it is written but a status register when it is read. In that case, you will need to keep track of the bit set in the functional register using a shadow variable, a global variable in your code that you modify whenever you change the register. Instead of setting and clearing intermediate registers, you will need to modify your shadow and then write the register with the shadow's value.

There are also registers where the act of reading the register modifies its contents. This is commonly used for status registers, where the pending status is cleared by the processor once your code reads the register (see

part C of Figure 5-8). This can prevent a race condition from occurring in the time between the user reading the register and clearing the relevant bit.

Your user manual will tell you which types of registers exhibit special behaviors.

Multiple sources for one interrupt

Some processors have only one interrupt. The machinery necessary to stop the processor takes up space in the silicon, and a single interrupt saves on cost (and power). When this interrupt happens, the ISR starts by determining which of the peripherals activated the interrupt. This information is usually stored in a *cause* or *status* register, where the ISR looks at each bit to separate out the source of the interrupt ("Timer 1, did you trigger an interrupt? No? What about you, timer 2?").

Having an interrupt for each possible peripheral is a luxury of many larger processors. When there are many possible sources, it is inefficient to poll all of the options. However, even when you have identified a peripheral, there is still a good chance you'll need to unravel the real source of the interrupt. For example, if your timer 3 is used for multiple purposes, the yellow light timeout may be indicated when timer 3 matches the first match register. To generate the appropriate event when the interrupt happens, you'll need to look at the peripheral's interrupt register to determine why the timer 3 interrupt occurred:

```
if (TIM2->SR & 0x1) { // interrupt occurs when timer's counter is updated
```

Most peripherals require this second level of checking for the interrupt. To look at another example, in a specific communication mechanism such as SPI, you'll probably get a single interrupt to indicate that something interesting happened. Then you'll need to check the SPI status register to determine what it was: it ran out of bytes to send, it received some bytes, the communication experienced errors, etc.

TIP

For GPIOs, there may be one interrupt for a bank of pins and you need to determine the specific pin that caused the interrupt.

Whether you have one interrupt with many sources or many interrupts with even more sources, don’t stop looking at the cause register when you find the first hit. There could be multiple causes. That leads us back to the question of priorities: which interrupt source will you handle first?

Disabling interrupts

Nested interrupts are allowed on the STM32F103xx, but we don’t want to deal with the complexity of interrupts within interrupts. The `__disable_irq()` and `__enable_irq()` macros come from the compiler vendor HAL and insert a single instruction, so the overhead to prevent nesting is minimal.

Critical sections

We’ve already seen how race conditions can happen in critical sections. To avoid that, we’ll need to turn off interrupts there as well. We could disable a particular interrupt, but that might lead to the priority inversion noted earlier. Unless you have a good reason for your particular design, it is usually safer to turn off all interrupts.

There are two methods for disabling interrupts. The first method uses the macros we’ve already seen. However, there is one problem with those: what if you have critical code inside critical code? For example:

```
HandyHelperFunction:
  disable interrupts
  do critical things
  enable interrupts

CriticalFunction:
  disable interrupts
  call HandyHelperFunction
  do supposedly critical things  // unprotected!
  enable interrupts
```

Note that as soon as interrupts are enabled in `HandyHelperFunction`, they are also enabled in the calling function (`CriticalFunction`), which is a bug. Critical sections should be short (to keep system latency at a minimum), so you could avoid this problem by not nesting critical sections, but this is something easier said than done.

NOTE

Because some processors won’t let you nest critical pieces of code, you’ll need to be sure to avoid doing it accidentally. To avoid this issue, I recommend naming the functions in a way that

indicates they turn off interrupts.

Alternatively, if your processor allows it, implement the global disable and enable functions (or macros) a little differently by returning the previous status of the interrupts in the code that disables them:

```
HandyHelperFunction:
  interrupt status  = disable interrupts()
  do critical things
  enable interrupts(interrupt status)

CriticalFunction:
  interrupt status = disable interrupts()
  call HandyHelperFunction
  do critical things
  enable interrupts(interrupt status)
```

Here the helper function gets an interrupt status indicating the interrupts are already off. When the helper function calls the method to enable the interrupts, the interrupt status parameter causes no action to be taken. The critical code remains safe throughout both functions. Also important, if the helper is called by a different function, the helper's critical area is still protected.

Restore the Context

After your ISR has finished, it is time to return to normal execution. Some compilers extend C/C++ to include an `interrupt` keyword (or `__IRQ`, or `_interrupt`) to indicate which functions implement interrupt handlers. The processor gives these functions special treatment, both when they start (some context is saved before the ISR starts running) and when they return.

As noted in the section on saving the context, the program counter points to the machine instruction you are about to run. When you call a function, the address of the next instruction (program counter + 1 instruction) is put on the stack as the return address. When you return from the function (`rts`), the program counter is set to that address.

NOTE

It is not unusual for different assembly languages to have similar opcodes. However, `rts` and `rti` tend to be pretty common. They stand for "ReTurn from Subroutine" and "ReTurn from Interrupt," respectively.

However, the interrupt isn't a standard function call; it is a jump to the interrupt handler caused by the processor. If the interrupt simply returned as though it were a function, the things that the processor did to store the context would not get undone. So interrupts have a special assembly instruction (`rti`) to indicate that they are returning from an interrupt. This lets the processor know that it must restore the context to its state prior to the function call before continuing on its way.

If your C/C++ compiler doesn't require you to indicate that a function is an interrupt, you can rest assured it is finding some other way to make the return from interrupt happen. That is, the compiler is probably wrapping the interrupt function in assembly code that merely calls your function. Once your handler returns from the function call, the assembly wrapper returns from the interrupt. The processor resets the stack the way it was, and program execution continues from exactly the point it left off.

Configuring Interrupts

The first step to setting up an interrupt is often, somewhat oddly, disabling the interrupt. Even though part of the power-on sequence is to disable and clear all interrupts, it is sensible to take the precaution anyway. If the interrupt is enabled, it might fire before the initialization code finishes setting it up properly, possibly leading to a crash.

Setting up interrupts uses registers that are memory mapped, similar to those we saw in "Function Pointers Aren't So Scary". As noted in that sidebar, accessing the memory address directly will make for illegible code. Most processor vendors and compiler vendors will give you a header file of `#define` statements, often allowing you to access individual registers as members of structures. Let's take apart a typical line of code:

```
NVIC->ICER[0] = (1<<2); // disable timer 2 interrupt
```

Many things are going on in that one line. First, `NVIC` is a pointer to a structure located at a particular address. The header file from the compiler vendor unravels the hard-coded memory-mapped address:

```
#define SCS_BASE  (0xE000E000UL)        /*!< System Control Space Base Addr */
#define NVIC_BASE (SCS_BASE + 0x0100UL) /*!< NVIC Base Address  */
#define NVIC   ((NVIC_Type *)NVIC_BASE) /*!< NVIC configuration struct */
```

The same header file defines the structure that holds the registers and where they are in the address space, so we can identify the next element in our line

of code, `ICER[0]`:

```
typedef struct{
  __IO uint32_t ISER[8]; /*!< Offset: 0x000  Interrupt Set Enable Register */
       uint32_t RESERVED0[24];
  __IO uint32_t ICER[8];/*!< Offset: 0x080  Interrupt Clear Enable Register*/
…
}  NVIC_Type;
```

NOTE

This header file structure can be built by reading the user manual if you can't find where someone else has done it for you.

This processor has separate clear and set registers, as described in "Writing an Indirect Register", so what our code does is set a bit in the clear (`ICER`) register that disables the timer interrupt. This is known as *masking* the interrupt.Once the interrupt is disabled, we can configure it to cause an IRQ when the timer has expired. (Timer configuration was covered in Chapter 4). All operations use the structure that points to the memory map of the processor registers for this peripheral (TIM2). The user manual says that the match control register describes whether an interrupt should happen (yes) and whether the timer should reset and start again (no):

```
TIM2->DIER |= 0x01;      // interrupt definition reg, on update
TIM2->CR1  |= TIM_OPMODE_SINGLE; // stop incrementing after it expires
```

Even though the first line of this snippet sets the timer interrupt to occur when it modified the register, it didn't really turn on the interrupt. Many processors require two steps to turn on the interrupt: a peripheral-specific interrupt-enable like the one just shown, plus a global interrupt-enable like the following:

```
NVIC->ISER[0] = (1<<2);        // enable timer 2 interrupt
TIM2->CR1  |= TIM_CR1_CEN;    // start the timer counting
```

Note that the peripheral interrupt configuration is in the peripheral part of the user manual, but the other register is in the interrupts section. You need to set both so that an interrupt can happen. Before putting that line in our code, we might want to configure a few more things for our timer interrupt.

When to Use Interrupts (and When Not To)

Now that we've set up our stoplight to use an interrupt for the timer and created the code to handle the interrupt, we need to backtrack. We forgot a

design step: should the yellow light timeout be an interrupt?

There are many circumstances in which the simplest solution is an interrupt. Communication pathways often have buffers that need to be filled (or emptied). An interrupt can act as a background task to feed the buffers while the foreground task generates (or uses) the data. Changes to input lines may need interrupts if they need to be handled quickly. The more real-time the requirement to handle a change on the line, the more likely an interrupt is appropriate for a solution.

Under that criterion, the button press we saw in Chapter 4 that needed only a simple response within 50 ms would not need an interrupt. A button press happens pretty rarely in the world of your processor. However, if checking to see whether it has been pressed takes time from other activities, it may be better to have an interrupt.

We've also seen that interrupts have some serious downsides. I already mentioned the overhead of each interrupt, which can add up if you have a lot of them. Interrupts also make your system less deterministic. One of the great things about not having an operating system[1] is being able to say that once instruction *x* happens, *y* will happen. If you have interrupts, you lose the predictability of your system. And because the code is no longer linear in flow, debugging is harder. Worse, some catastrophic bugs will be very difficult to track down if they depend on an interrupt happening at a very specific time in the code (i.e., a race condition). Plus, the configuration is largely compiler- and processor-dependent (and the implementation may be as well), so interrupts tend to make your code less portable.

In the end, the development cost of implementing (and maintaining) interrupts is pretty high, sometimes higher than figuring out how to solve the problem at hand without them. Save interrupts for when you need their special power: when a system is time-critical, when an event is expensive to check for and happens very rarely, and when a short background task will let the system run more smoothly.

How to Avoid Using Interrupts

So if you don't need their special power, how can you avoid interrupts? Some things can be solved in hardware, such as using a faster processor to keep up with time-critical events. Other things require more software finesse.

Returning to our stoplight example, we've considered the events as interrupts. Two events are communicated to the system (the commands stop and go), and one is generated by the system (yellow's timeout). Do any of these events need to be interrupts?

Let's assume the system doesn't do anything besides handle the events. It just waits for these events to happen. The implementation of the system could be much simpler to maintain if we can always see what the code is waiting for.

NOTE

When you get to Chapter 11, interrupts become a way to wake the processor from sleep, so you have to use them.

Polling

Asking a human "are you done yet?" is generally considered impolite after the fourth or fifth query, but the processor doesn't care if the code incessantly asks whether an event is ready. Polling adds processor overhead, even when there are no events to process. However, if you are going to be in a while loop anyway (i.e., an idle loop), there is no reason not to check for events.

Polling is straightforward to code. There's one subtlety worth mentioning: if you are polling and waiting for the hardware to complete something, you might want a timeout, just in case.

In the yellow light example, all we need to do is wait for a certain amount of time to pass. Even though embedded systems have a reputation for being fast, many systems spend an inordinate amount of their clock cycles waiting for time to pass.

System Tick

Like the sound you hear when the clock's second hand moves, many systems have a tick that indicates time is passing. The amount of time in that tick varies, but one millisecond tends to be a popular choice.

NOTE

Ticks don't have to be one millisecond. If you have a time that is important to your system for other reasons (e.g., you have an audio recording system that is running at 44,100 Hz), you might

want to use that instead.

The tick is implemented with a timer interrupt that counts time passing. Yes, if we implemented the yellow light timeout this way, we are still basing the solution on an interrupt, but it is a much less specific interrupt. The system tick solves a much broader range of problems. In particular, it lets us define this function:

```
void DelayMs(tTime delay);
```

This will wait for the amount of time indicated—well, for approximately the amount of time indicated (see the following sidebar, “Fenceposts and Jitter”). Note that because of fenceposting and jitter, `DelayMs` isn’t a good measure of a single millisecond. However, if you want to delay 10 or 100 milliseconds, the error becomes small enough not to matter. If your system depends on one-millisecond accuracy, you could use a shorter tick, though you’ll need to balance the overhead of the timer interrupt with the processing needs of the rest of the system.

FENCEPOSTS AND JITTER

A fencepost error is an example of an off-by-one error, one often illustrated with building materials:

If you build a straight fence 100 m long with posts 10 m apart, how many posts do you need?

The quick and wrong answer is that you need 10 posts, but you actually need 11 posts to enclose the 100 meters, as shown in Figure 5-9. The same is true of a system tick. To cover at least the number of milliseconds in question, you need to add one to the delay.

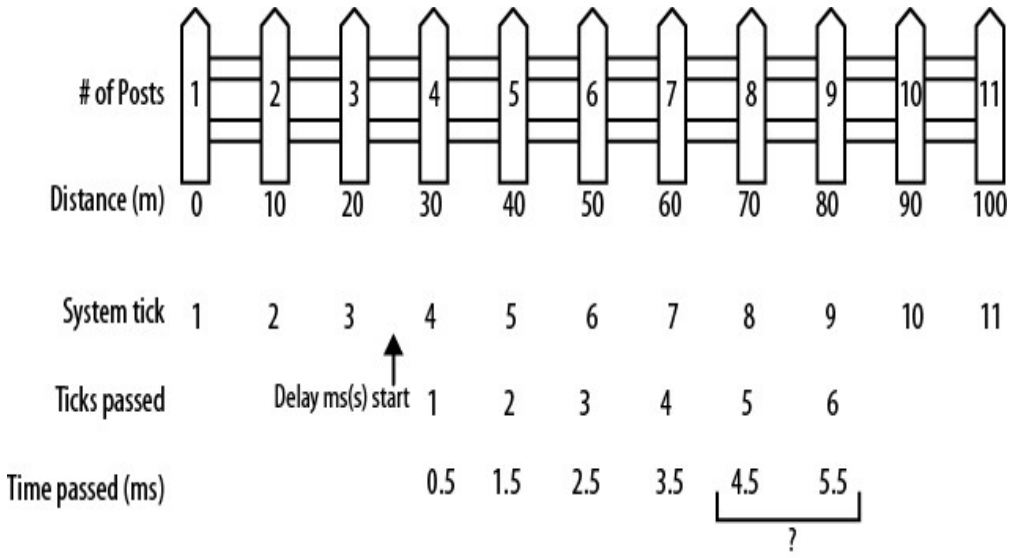


Figure 5-9. Fencepost example

However, this calculation is complicated by *jitter*. Your call to `DelayMs` is very unlikely to happen on a tick boundary. Instead, the delay will always start after a tick, so that `DelayMs` will always be longer than the number of ticks indicated.

You can choose whether you want to wait no more than the delay indicated (in which case, with a one millisecond delay, you may wait less than a processor cycle) or no less than the delay indicated (with a one millisecond delay, you may wait two milliseconds minus a processor cycle). You can make whatever choice leaves your application most robust, as long as the code is clear.

In the stoplight's yellow state, we could call `DelayMs` and move to red as soon as it finishes. However, then we couldn't respond to any other commands. In this example it happens to be OK because stop and go don't do anything in the yellow state, but still, what if one of them did?

If you want to keep track of time passing and do other things, add a few more functions to your system tick:

```
tTime TimeNow();
tTime TimePassed(tTime since);
```

The `TimeNow` function should return the tick counter. The code never looks at this directly, instead using the `TimePassed` function to determine whether enough time has passed. In fact, `DelayMs` can be implemented as a combination of these functions (getting the initial time and then waiting for

the time passed to be greater than the delay). In between these functions, your system can do other things. In effect, this is a kind of polling.

Note that the `TimeNow` function gives the number of ticks since the system was booted. At some point this variable holding the number of ticks will run out of space and roll over to zero. If you used an unsigned 16-bit integer, a 1 ms tick would make the clock roll over every 65.5 seconds. If you need to use these functions to try to measure something that takes 70 seconds, you may never get there.

However, if you use an unsigned 32-bit integer, your system will roll over to zero in 4,294,967,296 ms, or about 49.7 days. If you use an unsigned 64-bit integer, your 1 ms tick won't roll over for half an eon (0.58 billion years). I'm impressed by this long-term thinking, but are you sure there won't be a power outage or system reboot before then?

So the size of your timekeeping variable determines the length of time you can measure. In many systems, instability can occur when the rollover happens. Protect against this by taking the rollover into account when you write the time-measuring function, to ensure the discontinuity does not cause a problem:

```
tTime TimePassed(tTime since) {
  tTime now = gSystemTicks;

  if (now >= since) {return (now - since);}

  // rollover has occurred
  return (now + (1 + TIME_MAX - since));
}
```

Time-Based Events

In our stoplight example, the yellow state can use the system tick to create its time-based event. When the code enters the yellow state, it sets a state variable:

```
yellowTimeStart = TimeNow();
```

Then, when doing housekeeping in the yellow state, it checks for the completion of the event:

```
if (TimeSince(yellowTimeStart) > YELLOW_STATE_TIMEOUT)
  // transition out of the yellow state
```

Between those two times, the system can do whatever it needs to: listen for commands, check the lights to make sure their bulbs are working, play Tetris, etc.

A Very Small Scheduler

For things that recur or need attention on a regular basis, you can use a timer as a mini-scheduler to fire off a callback function (a task).

NOTE

At this point, you are starting to re-create the functions of an operating system. If your mini-scheduler becomes more and more complicated, consider investing in an operating system.

Let's add a state to our stoplight that blinks the red light to indicate a four-way stop. This happens only when something goes wrong, but we'll cover those possibilities in the next section. Although we could use the time-based event to turn the red light on and off, it might be a little simpler to make this a background task, something that the scheduler can accomplish.

So, before looking at the gory details, let's consider what the main loop needs to do to make all this work for a scheduler that runs about once a second, appropriate for our red-light blinking state (but too sluggish for handling other events):

```
  Run the scheduler after initialization is complete
  LastScheduleTime = TimeNow()

  Loop forever:
    if TimePassed(LastScheduleTime) > ONE_SECOND
     Run the scheduler, it will call the functions (tasks) when they are due
     LastScheduleTime = TimeNow()
```

We already worked out the time-based functionality in the previous section; now let's look at the scheduler in more detail.

When a task is allocated, it has some associated overhead, including the callback function that is the heart of the task. It also contains the next time the task should be run, the period (if it is a periodic task), and whether or not the task is currently enabled:

```
struct Task;                                // forward declaration of the struct
typedef void (*TaskCallback)(struct Task *); // type of the callback function

typedef struct Task {
  tTime runNextAt;              // next timer tick at which to run this task
  tTime timeBetweenRuns;        // for periodic tasks
  TaskCallback  callback;
  int enabled;                  // current status
};
```

The calling code should not know about the internals of the task management, so each task has an interface to hide those details.

```
void TaskResetPeriodic(struct Task *t);
void TaskSetNextTime(struct Task *t, tTime timeFromNow, tTime now);
void TaskDisable(struct Task *t);
```

(Yes, these could be methods in a class instead of functions; both ways work fine.)

The scheduler is straightforward and has only one main interface:

```
void SchedulerRun(tTime now);
```

When the scheduler runs, it looks through its list of tasks. Upon finding an enabled tasks it looks to see whether the current time is later than `runNextAt`. If so, it runs the task by calling the callback function. The `SchedulerRun` function sits in the main loop, running whenever nothing else is.

These tasks are not interrupt-level, so they should not be used for activities with real-time constraints. Also, the tasks have to be polite and give up control relatively quickly, because they will not be preempted as they would be in typical operating systems.

There is one final piece to the scheduler: attaching the tasks to the scheduler so that they run. First, you'll need to allocate a task. Next, configure it with your callback function, the time at which the function should run, and the time between each subsequent run (for periodic functions). Finally, send it to the scheduler to put in its list:

```
void SchedulerAddTask(struct Task* t);
```

PUBLISH/SUBSCRIBE PATTERN

With the scheduler, we've built what is known as a publish/subscribe pattern (also called an observer pattern or pub/sub model). The scheduler publishes the amount of time that has passed, and several tasks subscribe to that information (at varying intervals). This pattern can be even more flexible, often publishing several different kinds of information.

The name of the pattern comes from newspapers, probably the easiest way to remember it. One part of your code publishes information, and other parts of your code subscribe to it. Sometimes the subscribers request only a subset of the information (like getting the Sunday edition only). The publisher is only loosely coupled to the subscribers. It

doesn’t need to know about the individual subscribers; it just sends the information in a generic method.

Our scheduler has only one type of data (how much time has passed), but the publish/subscribe pattern is even more powerful when you have multiple types of data. This pattern is particularly useful for message passing, allowing parts of your system to receive messages they are interested in but not others. When you have one object with access to information that many others want to know about, consider the publish/subscribe pattern as a good solution.

Watchdog

We started the scheduler section mentioning the blink-red state. Its goal was to put the system in a safe mode when a system failure occurs. But how do we know a system failure has occurred?

Our software can monitor how long it has been since the last communication. If it doesn’t see a stop or go command in an unreasonably long time, it can respond accordingly. Metaphorically speaking, it acts as a watchdog to prevent catastrophic failure. Actually, the term *watchdog* means something far more specific in the embedded world.

Most processors or reset circuits have a watchdog timer capability that will reset the processor if the processor fails to perform an action (such as toggle an I/O line or write to a particular register). The watchdog system waits for the processor to send a signal that things are going well. If such a signal fails to occur in a reasonable (often configurable) amount of time, the watchdog will cause the processor to reset.

The goal is that when the system fails, it fails in a safe manner (failsafe). No one wants the system to fail, but we have to be realistic. Software crashes. Even safety-critical software crashes. As we design and develop our systems, we work to avoid crashes. But unless you are omniscient, your software will fail in an unexpected way. Cosmic rays and loose wires happen, and many embedded systems need to be self-reliant. They can’t wait for someone to reboot the system when the software hangs. They might not even be monitored by a human. If the system can’t recover from some kinds of error, it is generally better to restart and put the system in a good state.

Using a watchdog does not free you from handling normal errors; instead, it exists only for when the system is unrecoverable. There are ways to use a watchdog that make it more effective. But first, let's look at some suboptimal techniques, based on models earlier in this chapter, that would make the watchdog less effective:

Setting up a timer interrupt that goes off slightly more often than the watchdog would take to expire

> If you service the watchdog in the timer interrupt, your system will never reset, even when your system is stuck in an infinite loop. This defeats the purpose of the watchdog.

Setting the delay function (`DelayMs`*) to service the watchdog, with the idea that the processor isn't doing anything else then*

> You'll have delay functions scattered around the code, so the watchdog will get serviced often. Even then, if the processor gets stuck in an area of code that happens to have a delay, the system won't reset as it should.

Putting the watchdog servicing in places that take a while for the processor to perform, maybe the five or six longest-running functions

> By scattering the watchdog code around, it waters down the power of the watchdog and offers the possibility that your code could crash in one of those areas and hang the system.

The goal of the watchdog is to provide a way to determine whether any part of the system has crashed. Ideally, watchdog servicing is in only one place, some place that the code has to pass through that shows all of its subsystems are running as expected. Generally, this is the main loop. Even if it means that your watchdog needs to have a longer timeout, having it watch the whole system is better than giving it a shorter recovery time while trying to watch only part of the system.

Sadly, for some systems, the watchdog cannot be segregated so neatly. When the signal to the watchdog must be sent in some lower level of code, recognize that the code is dangerous, an area where an unrecoverable error

might occur, causing the system to hang. That code will need extra attention to determine whether anything could go wrong (and hopefully prevent it).

Generally, you don't want the watchdog active during board bring-up or while using a debugger. Otherwise, the system will reset after a breakpoint. A straightforward way to turn off the watchdog will facilitate debugging. If you have a logging method, be sure to print out a message on boot if the watchdog is on. It is one of those things you don't want to forget to enable as you do production testing. Alternatively, you can toggle an LED when the watchdog is serviced to give your system a heartbeat that is easy to see from the outside, letting the user know that everything is working as expected.

WATCHDOG TERMINOLOGY

Servicing the watchdog so it doesn't hang has traditionally been called "kicking the dog."

This caused consternation when I worked at a small pet-friendly company. It was great to see the dogs play at lunch, and kind of calming. However, one day as we were preparing for a big client to review our code, our CEO went off the rails when he saw the function `KickTheDog()` in the main loop. He adamantly explained that no dog, real or virtual, would be kicked at our company. Ever.

We refactored the function name to `PetTheDog()`. On the day of the big meeting, the clients snickered when they saw our politically correct safety net. I'm not sure when the terminology changed, but in the years since then, I've seen less kicking and more petting, feeding, or walking the watchdog. What you choose to do is up to you, but you'd never actually kick a dog, right?

There are three takeaways here: 1) terminology for standard things changes, 2) language matters, and 3) never let your CEO see your code.

Main Loops

This chapter started with a few important notes about RTOS terms (even though I'm not supposing you have an RTOS on your system, the concepts transfer to bare metal). Then there were a bunch of ways to put together state machines. Finally, I nattered on about interrupts until you thought about

skipping pages. Then we went back to RTOS ideas with a very small scheduler. This is all well and good but how do we really get started?

We usually start in the main loop. As with state machines, there are many ways to set up a main loop. They all have advantages and disadvantages so you'll need to figure out what is best for you.

Polling and Waiting

I like Arduino's simple and clear program structure.

```
setup:
  init
loop:
  toggle LED
  delay
```

Behind the scenes, the Arduino development environment puts in a main function that calls the setup function to do initialization. The hidden main function does other board and processor initialization that it needs. Once your setup and the hidden initialization is complete, it starts a forever loop, calling your loop function on each pass (but also possibly calling its own set functions as needed). Figure 5-10 shows how the pieces of the system interact. The development environment hides the details from you but somewhere, there is a main function.

We don't need someone else to do this for us, we can make `main` ourselves:

```
main:
  init
  while (1)
    toggle LED
    delay
```

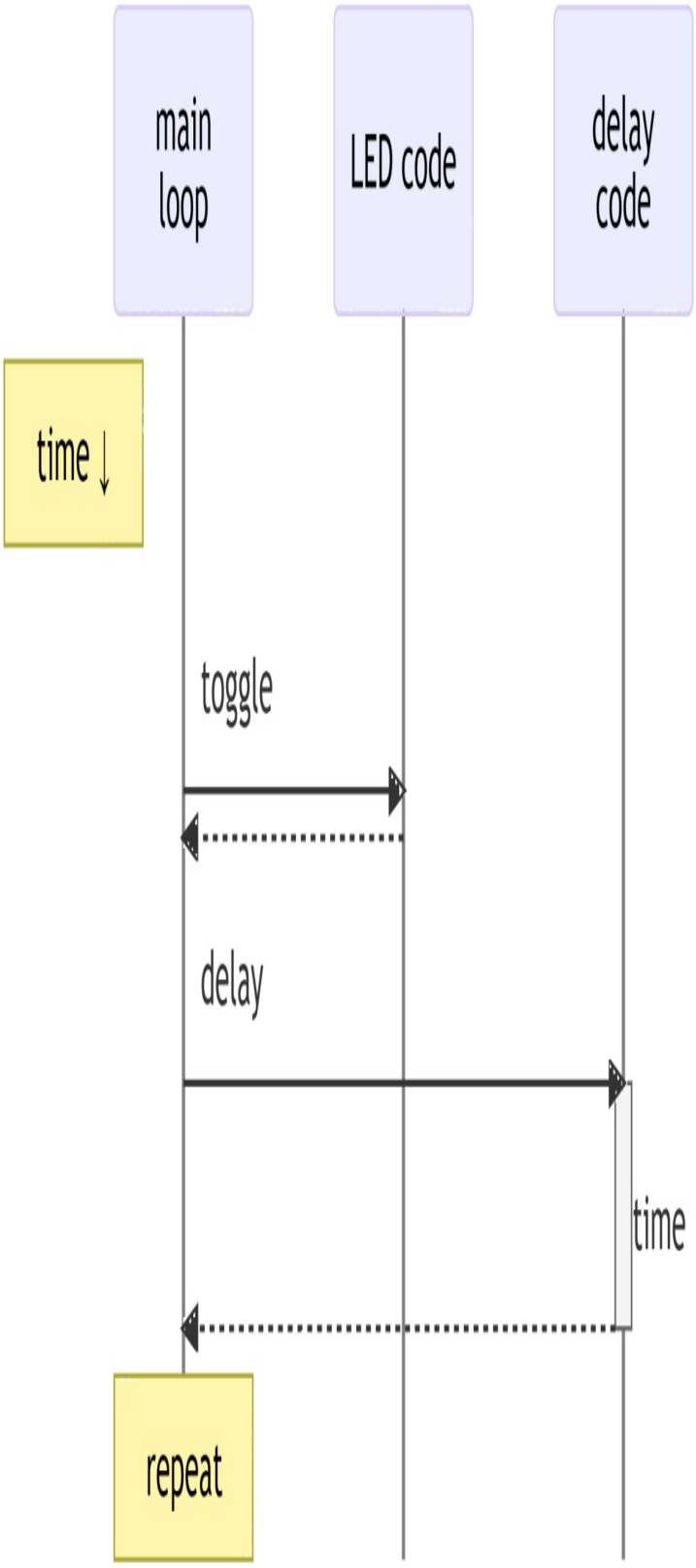

main loop
LED code
delay code
time ↓
toggle
delay
time
repeat

Figure 5-10. Simplest main loop: toggles an LED

This is how main loops begin their lives, as something simple. Of course, this blinking light is extremely boring, so I'm going to need to add something.

```
main:
  init
  while (1)
    read inputs
    write outputs
    toggle LED
    delay
```

The read inputs and write outputs could mean anything; it could be where you have the function to check on your command handler interface or where you update your state machine based on the current time, polled buttons, or the whatever inputs you have to create whatever outputs you need. See Figure 5-11 for the sequence of how things happen in this loop.

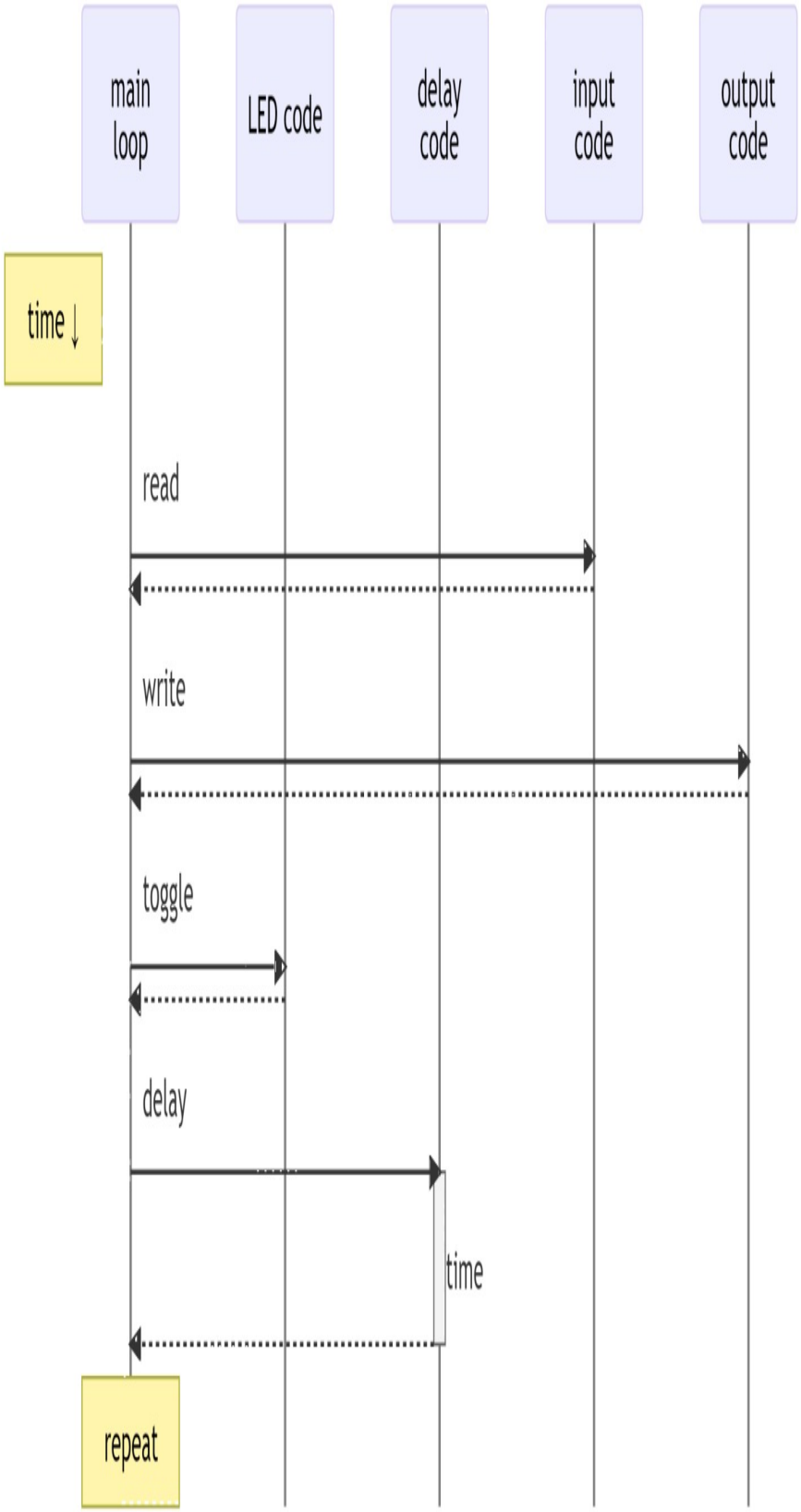

main loop
LED code
delay code
input code
output code
time ↓
read
write
toggle
delay
time
repeat

Figure 5-11. Blocking main loop

Everything happens as expected in this loop but there is a big problem: it is blocking. If inputs come in while the system is running the delay, they won't get handled until the next pass of the loop. If the inputs need to be dealt with quickly, this may not work.

Timer Interrupt

Blocking the processor with a delay is silly when a timer interrupt could do the work toggling the LED.

```
timer interrupt: toggle LED
main:
  init
  while (1)
      read inputs
    write outputs
    pet watchdog
```

Toggling an IO pin takes very little time, it is safe to put it in an interrupt.

One downside is that the LED is no longer connected to reading and writing, it tells us only that the timer interrupt is firing off at regular intervals. However, I added a watchdog to make sure the loop stays running as expected. The sequence diagram for this (Figure 5-12) shows the LED code separated from the other pieces of code. The increased system complexity is balanced with decreased local complexity (main doesn't need to pay attention to the LED).

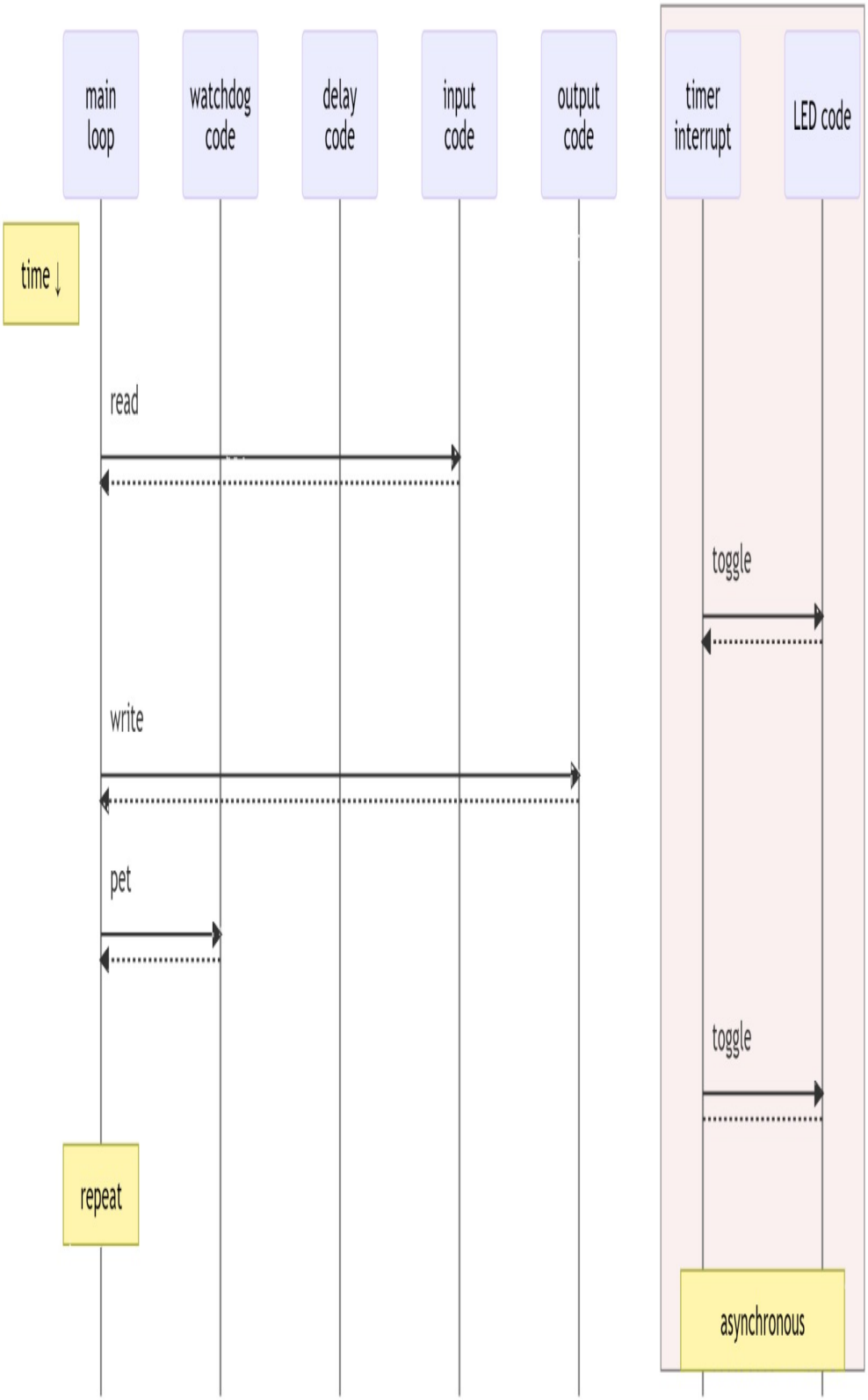

main loop
watchdog code
delay code
input code
output code
timer interrupt
LED code
time
read
toggle
write
pet
toggle
repeat
asynchronous

Figure 5-12. Main loop with timer interrupt controlling LED

This method assumes that the read input and write outputs are quick; that are not blocking each other and that they each happen quickly enough not to interfere with the timing of the other.

Interrupts Do Everything

But what if they can't? What if sometimes the input needs to run immediately and can't wait for the output to finish doing its thing?

Assuming there is a way to set an interrupt when the input data is available, we can have an interrupt handle copying the data in. This may be a sensor that sends an interrupt to start sending data or a communication pathway that lets you set up an interrupt when its FIFO is half full. On the output side, perhaps there is a communications pathway that takes time but will interrupt to tell you when it is ready for the next message. So what if the interrupts do all the work?

```
timer interrupt: toggle LED
data available interrupt: copy data in
output ready interrupt: copy data out

main:
  init
  set up interrupts
  while (1)
    pet watchdog
    sleep
```

There is a lot of goodness to this. The decoupling of the subsystems is good as it leads to a separation of concerns with every interrupt doing their own thing, not depending on the others. Also, with the interrupts indicating when things happen, the main loop can go to sleep, awoken by the interrupts. This behavior is great for power reasons (more about that in Chapter 10).

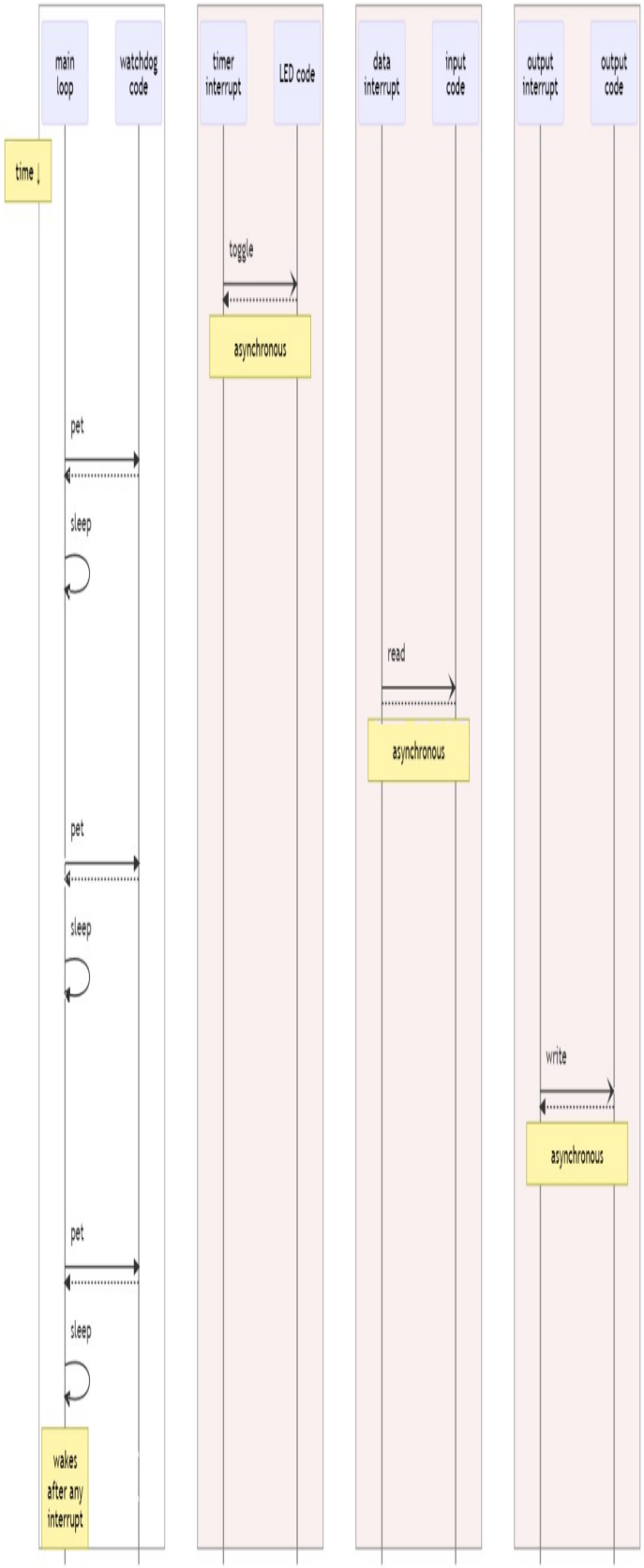

main loop
watchdog code
timer interrupt
LED code
data interrupt
input code
output interrupt
output code
time ↓
toggle
asynchronous
pet
sleep
read
asynchronous
pet
sleep
write
asynchronous
pet
sleep
wakes after any interrupt

Figure 5-13. Main loop where the interrupts do all the work

As the sequence diagram in Figure 5-13 shows, each of these pieces of code are asynchronous, happening whenever they happen, not dependent on each other. Well, mostly not dependent. If an interrupt wakes the processor, the ISR runs, then returns to the main loop, directly after the sleep call. In my case, it pets the watchdog and goes back to sleep. The downside of asynchronicity is that understanding the flow of execution is much harder (which makes debugging much more difficult).

Because the interrupts happen when they need to, their response time (system latency) should be fast, as long as another interrupt isn't blocking them from running. Priority and missed interrupts can become issues if interrupts take too long.

Of course, interrupts are supposed to be short. Copying a bunch of data is usually not a short process; in fact, it is the sort of thing we want to avoid doing in an interrupt service routine.

Interrupts Cause Events

Instead, we can set a variable that tells main what to do, keeping our interrupts short, allowing the processor to sleep, and keeping the system responsive to the current needs.

```
timer interrupt: set global led_change variable
data available interrupt: set global data_available
output ready interrupt: set global output_ready

main:
  init
  while (1)
    if led_change: toggle led; led_change = false
    if data_available: copy input; data_available = false
    if output_ready: copy output; output_ready = false
    pet watchdog
    sleep
```

This adds a bunch of global variables to the code, often a good way to create tangled spaghetti code. However, the ordering of events is less uncertain as we can use the global variables to control ordering (only run `output_ready` after `data_available`). The interrupts are still asynchronous but as the logic is in normal execution, we can add a few logging calls which will make debugging much easier.

Plus, the processor can still sleep as the interrupts will wake it up.

Not shown in the pseudocode are two important points I've already mentioned earlier in the book. First, the global variables are volatile so they don't get removed by the compiler's optimizer (because according to the compiler's view of the main loop, nothing ever changes about event variables). Second, the global variables are only modified with the interrupts disabled in order to prevent race conditions.

And if that code with all of its if statements looks like a state machine to you, well, thank you for paying attention.

Figure-15 shows the events being created by the interrupts and handled in the main loop. Note that the interrupts are asynchronous so main will handle whatever event happens next, not in any particular order.

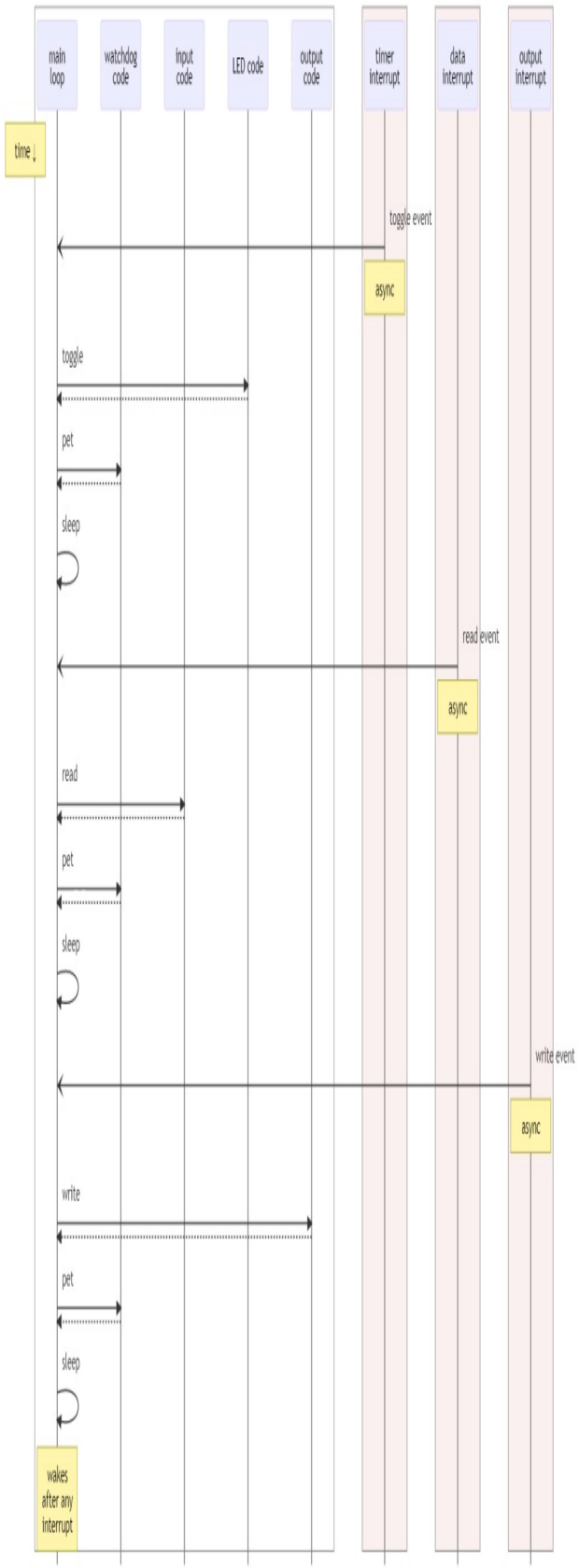

main loop
watchdog code
input code
LED code
output code
timer interrupt
data interrupt
output interrupt
time
toggle event
async
toggle
pet
sleep
read event
async
read
pet
sleep
write event
async
write
pet
sleep
wakes after any interrupt

Figure 5-14. Interrupts signal events that are handled in the main loop

Tiny Scheduler

What about the tiny scheduler? How does that look in this form? As noted above, it looks through a list of tasks, calling their associated callback function when it is time for that task to run. It is entirely time-based.

```
systick interrupt: increment current time

main:
  init
  while (1)
    for each task
      if time to run task
         call task callback function
        set new time to run task
    pet watchdog
```

With this method, each task *runs to completion*. It is a function call so it may be interrupted by an interrupt (like the system timer tick) but not by any other task. It will finish before the next task can run. That also means if it takes a long time, functions scheduled after it can be delayed.

If this seems totally normal to you, I agree but an RTOS can be *preemptive* which means that a more important task can interrupt less important tasks to use the processor. The RTOS's scheduler usually looks for the opportunity to do this anytime an RTOS related function is called.

But here with the tiny scheduler, it runs in main and doesn't do anything fancy. When a task is done, it returns from its function. Then the processor can do other things like go on to the next task that is ready to run.

The processor can also sleep, as long as it can schedule a wake up interrupt when the next task will be ready to run. Most timers allow this sort of thing but not all. If not, the system will have to wake up on each new systick interrupt.

The sequence diagram in Figure 5-15 shows this in action. Note that the tasks are asynchronous with respect to each other (they don't need to go in the order shown in the diagram). Instead, each task depends on how often it needs to run.

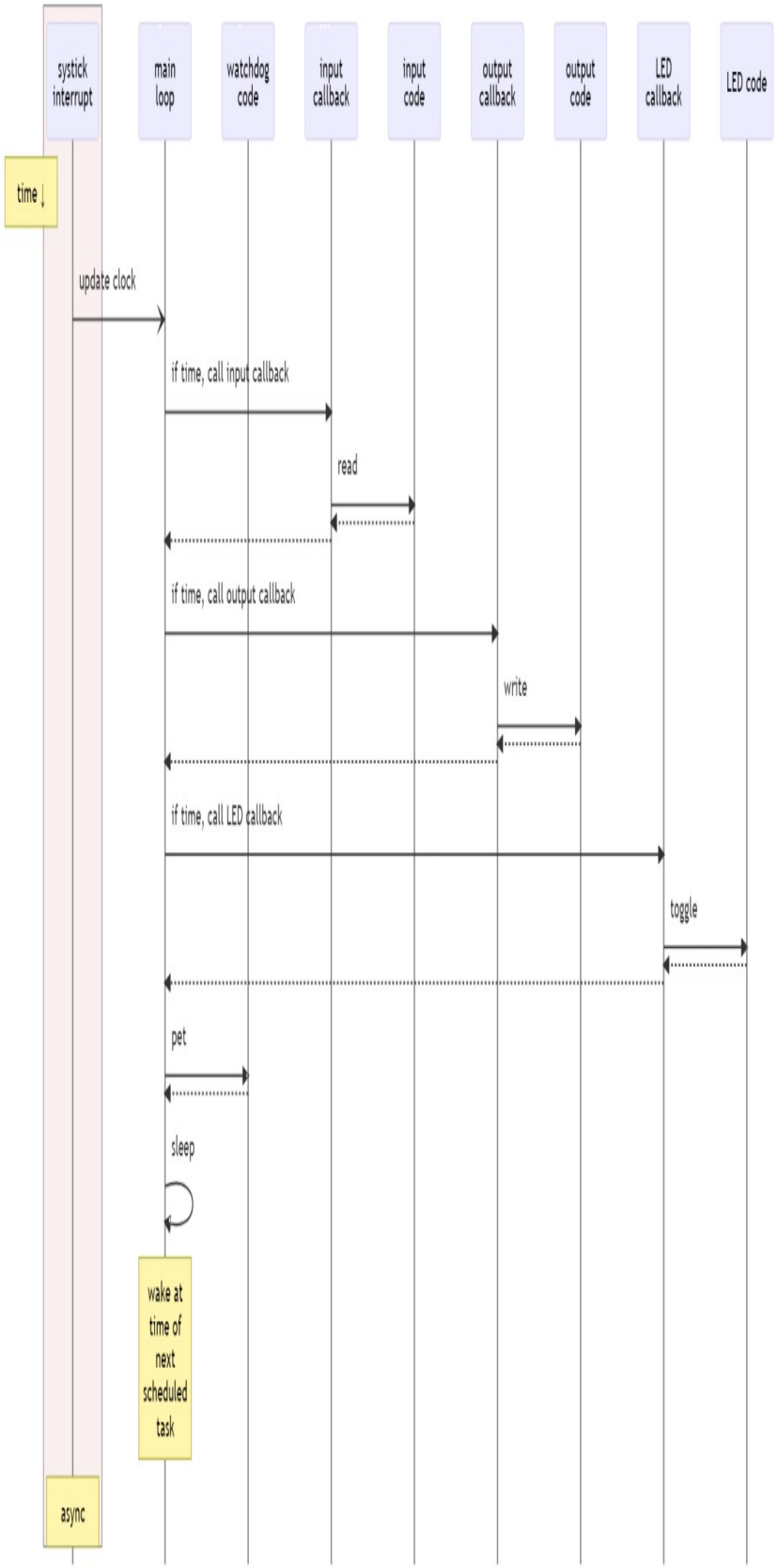

systick interrupt
main loop
watchdog code
input callback
input code
output callback
output code
LED callback
LED code
time ↓
update clock
if time, call input callback
read
if time, call output callback
write
if time, call LED callback
toggle
pet
sleep
wake at time of next scheduled task
async

Figure 5-15. Tiny scheduler drives all tasks, running them at given times

Not everything has to go through the scheduler, you can have other interrupts. For example, a data available interrupt may change the list of tasks so that the inputCallback happens immediately (well, the next time the scheduler checks if that task is ready to run).

Additionally, the LED could still be toggled by a separate timer interrupt. There is no hard and fast rule for any of these main loop styles; mashups are allowed.

Active Objects

There is one more style of main loop that I'd like to share. If you combine the synchronous (non-interrupt-y) simplicity of the tiny scheduler with the lovely separation of concerns when everything is done in interrupts, you get *active objects*. This one needs an RTOS to be effective because it needs truly separate tasks. However, I see it a lot in embedded systems from vendor demo code that uses Bluetooth or a state machine; code where you need to add your functions and tasks to an existing system.

The goal is to spend very little time in interrupts, let everything be completely asynchronous, and to make each piece of code as separate as possible. That way your code won't interfere with whatever system you are integrating into.

A key characteristic of active objects is *inversion of control*. Instead of your code controlling the system, the framework is in charge and will call you when it needs you.

```
timer interrupt: send event to timer task
timer task:
  while (1)
    wait for event message
    if message is toggle LED,  toggle LED

data available interrupt: send event to data task
data task:
  while (1)
    wait for event message
    if message is data available,
      handle data
      send output ready to output task

output task:
while (1)
    wait for message
    if message is output ready,
      output data
```

```
main:
  init
  while (1)
    pet watchdog
    sleep
```

While more complicated on surface, each task is completely devoted to what it does. Each task keeps its data private: they only communicate via message queues that are filled asynchronously. They try to remain autonomous from each other. The first few times I saw active objects it seemed like a lot of repetition, having to dig deeply through vendor code to trace how event messages travel through multiple tasks to be handled by the correct ones. However, looking at it from the vendor's perspective, the events are a sort of contract, telling you what is going on without knowing anything about your code.

TIP

Inversion of control is often referred to as the Hollywood Principle: Don't call us, we'll call you.

One good way to identify an active object is to see if the task is organized around a *message pump*. Usually this is implemented by a `wait for message` area followed by a large switch statement that depends on the message event received. Once the event is handled, the task goes back to waiting. The task should not do any blocking calls, the only place it waits is in that one spot (`wait for message`). (Not all code follows this rule, some systems allow delays or other RTOS calls to give up control. But it is good practice to have the message pump be the sole single blocking point.)

Figure 5-16 shows how separate each component of the system is, minimizing dependencies and only allowing specific methods of communication between tasks.

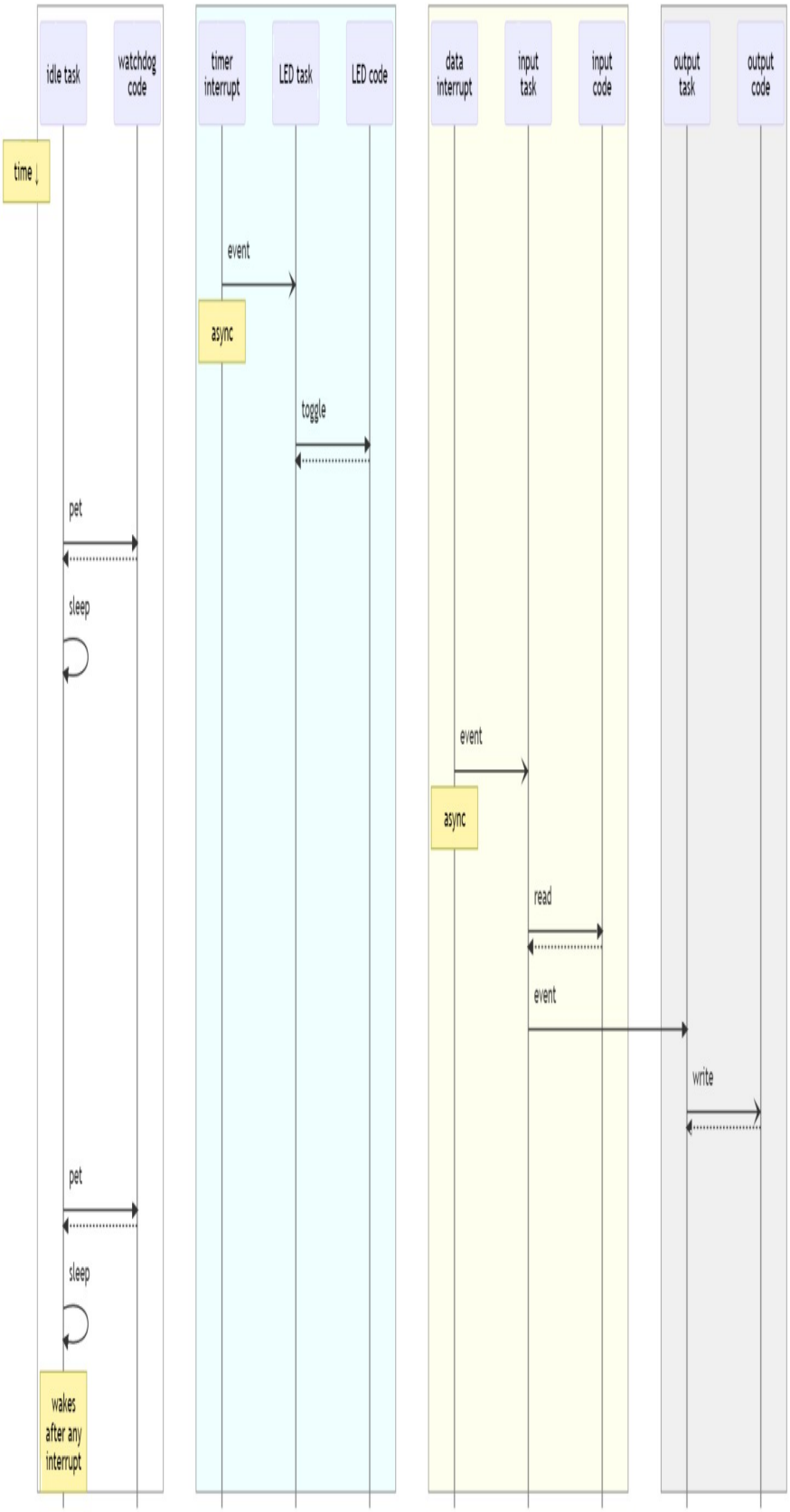

idle task
watchdog code
timer interrupt
LED task
LED code
data interrupt
input task
input code
output task
output code
time
event
async
toggle
pet
sleep
event
async
read
event
write
pet
sleep
wakes after any interrupt

Figure 5-16. Active objects run as true tasks in an operating system, using a message pump to separate dependencies between tasks.

Another name for an active object is the actor design pattern. These active objects are usually implemented as event-driven state machines. And if you recall dependency injection from Chapter 3, it shares a lot of the same philosophy as inversion of control. I want to say the terminology doesn't matter, the principles are the important part but the jargon makes it easier to discuss the principles.

You might think all this technobabble is overwhelming and not that important, but you never know when you might find yourself in a Star Trek episode where the Enterprise must be saved by a carefully designed active object.

Further Reading

Running an embedded system without an operating system (running on bare metal) doesn't mean you can be ignorant of operating systems principles. If anything, because you are doing the work yourself, you need to know more about how OSs function so you can re-create the parts you need. There are many good OS books, but my favorite is the classic text book: *Operating Systems: Design and Implementation* by Andrew Tannenbaum (Pearson). For a free resource, check out the OSDev Wiki (osdev.org).

I also recommend finding an old book about programming a small processor, something published in 1980 or before. For example, I have *Programming the 6502*, Third Edition, by Rodnay Zaks (Sybex). You can find one in a library or used bookstore. This sort of book takes out the modern fluff surrounding processors (the fluff that makes them easier to use) and provides insight into how the processors worked when they were simpler. Also, they make for pretty entertaining reading because the assumed knowledge is much different from current user manuals (mine starts with a section called "What is programming").

Embedded software engineers sometimes focus more on the hardware details that we need to learn or the algorithms associated with application of the device. Good software techniques aren't new. One of the best introductions to good software practices is *Designing Reusable Classes* by Ralph Johnson and Brian Foote. It is free online if you want to take a look.

Miro Samek of Quantum Leaps is an enthusiastic advocate for the active object pattern and good software in embedded systems (see the Resources section of state-machine.com). His books and blog are filled with excellent information but his YouTube videos are particularly instructive as he codes the systems as he talks.

INTERVIEW QUESTION: STOPLIGHT CONTROL OF AN INTERSECTION

A small city has decided their intersection is too busy for a stop sign, and they've decided to upgrade to a light. They've asked you to write the code for the light. There are four lights, each with a red, yellow, and green bulb. There are also four car sensors that can tell when a vehicle is stopped at the light. Where do you start? Tell me about your design, and then write some pseudocode. (During this time, I've drawn an intersection like the one shown in Figure 5-17. I tend to draw this intersection on their piece of paper if I can.)

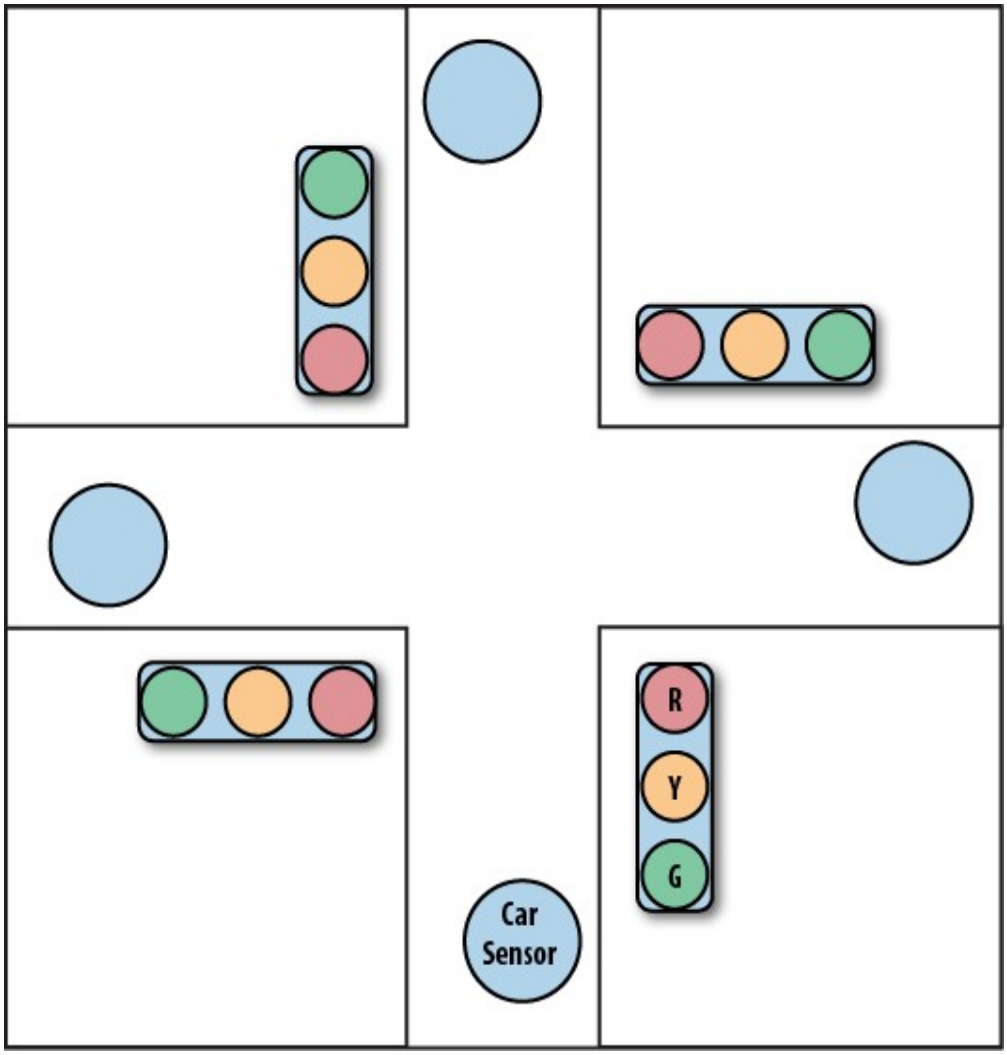


Figure 5-17. Intersection in a small city

This is a problem that lets the interviewee drive the interaction. If she wants to talk about design patterns, there are plenty. If she wants to skip design and talk about timers or the hardware of the sensing, that works too.

An interviewee should clarify the problem if it isn't clear. In this question, she should ask whether there is a left turn light (no, the intersection is small, and the city doesn't need that yet). Some people also ask about a crosswalk (also not to be handled in the initial development).

Very good interviewees notice that the problem is only half of what it appears on the surface. The goal is not to control four stoplights, since two lights are always in sync.

As with all of my interview questions, naming is very important. The reason I draw on their paper is to encourage the interviewee to add her own information to the diagram. Some good names include identifying the intersection by compass directions (north/south and east/west), and some moniker that makes sense for the situation (First and Main) or place on the page (right/left and up/down). Until she has put names on them, any pseudocode will be gibberish.

Once she starts digging into the problem, I like to hear about the state machine (or automata, or flow chart). I want to see diagrams or flowcharts. Some people get stuck on the initial state because we all tend to want to start with "what happens first." However, the initial state is relatively uninteresting in this case (though, if asked, I'll suggest she start with an all-red state).

Once she's laid out the basics, I tend to add a few curveballs (though great interviewees tend to handle these before I ask).

First, what if a sensor is broken? Or what if the city wants to be friendly to bikes (which don't activate the car sensor)? This adds a timer to the state machine so that the intersection doesn't get trapped in any state forever.

Second, the intersection has been getting a lot of accidents with folks running the yellow light. Can she do something to improve the safety of the system? (Hint: Add a short all-red state to allow traffic to clear the intersection before going to the next green.)

Because this interview question is simply a logic problem, I often spring it on other engineers, particularly those who work in quality departments ("How would you write a test plan for this controller?").

[1] Some real-time operating systems (RTOSs) are deterministic, usually the more expensive ones.

About the Author

Elecia is a senior embedded systems consultant at Logical Elegance Inc, in this role she has helped ship many consumer, medical, and industrial products. She is also the author of *Making Embedded Systems*, published by O’Reilly Media, and cohost of the Embedded.FM podcast and blog.

Table of Contents