
Making Embedded Systems

?? EDITION

Making Embedded Systems

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Making Embedded Systems
by

Printing History:

ISBN: 978-1-449-30214-6
1310483910

Table of Contents

Preface	xi
 1. Introduction	 1
Compilers, Languages, and Object-Oriented Programming	1
Embedded System Development	2
Debugging	2
More Challenges	4
Principles to Confront those Challenges	5
Further Reading	7
 2. Creating a System Architecture	 9
Creating System Diagrams	10
The Block Diagram	10
Hierarchy of Control	13
Layered View	15
From Diagram to Architecture	16
Encapsulate Modules	16
Delegation of Tasks	17
Driver Interface: Open, Close, Read, Write, IOCTL	18
Adapter Pattern	19
Getting Started With Other Interfaces	22
Example: A Logging Interface	22
A Sandbox to Play In	28
Further Reading	33
 3. Getting the Code Working	 35
Hardware/Software Integration	35
Ideal Project Flow	36
Board Bring Up	37
Reading a Datasheet	38
Datasheet Sections You Need When Things Go Wrong	40

Important Text for Software Developers	42
Evaluating Components Using the Datasheet	46
Your Processor Is a Language	49
Reading a Schematic	51
Having a Debugging Toolbox (and a Fire Extinguisher)	54
Keep your Board Safe	54
Toolbox	54
Digital Multimeter	55
Oscilloscopes and Logic Analyzers	56
Testing the Hardware (and Software)	59
Building Tests	60
Flash Test Example	61
Command and Response	64
Command Pattern	67
Handling Errors Gracefully	70
Consistent Methodology	70
Error Handling Library	71
Further Reading	71
4. Outputs, Inputs, and Timers	75
Toggling an Output	75
Starting with Registers	76
Set the Pin to be an Output	77
Turn On the LED	79
Blinking the LED	80
Troubleshooting	81
Separating the Hardware from the Action	82
Board-Specific Header File	82
I/O Handling Code	83
Main Loop	86
Facade Pattern	86
The Input In I/O	87
A Simple Interface to a Button	89
Momentary Button Press	91
Interrupt on a Button Press	92
Configuring the Interrupt	92
Debouncing Switches	93
Runtime Uncertainty	96
Dependency Injection	96
Using a Timer	98
Timer Pieces	98
Doing the Math	100
A Long Wait between Timer Ticks	105

Using the Timer	106
Using Pulse Width Modulation (PWM)	106
Shipping the Product	108
Further Reading	110
5. Task Management	111
Scheduling and Operating System Basics	111
Tasks	111
Communication between Tasks	112
Avoiding Race Conditions	114
Priority Inversion	115
State Machines	116
State Machine Example: Stoplight Controller	117
State-centric State Machine	117
State-Centric State Machine with Hidden Transitions	118
Event-centric	119
State Pattern	120
Table Driven State Machine	121
Choosing a State Machine Implementation	123
Interrupts	123
An IRQ Happens	124
Save the Context	129
Get the ISR from the Vector Table	131
Calling the ISR	133
Restore the Context	136
When To Use Interrupts	137
How Not to Use Interrupts	138
Polling	138
System Tick	138
Time Based Events	141
A Very Small Scheduler	141
Watchdog	143
Further Reading	145
6. Communicating with Peripherals	149
The Wide Reach of Peripherals	149
External Memory	149
Buttons and Key Matrices	150
Sensors	152
Actuators	155
Displays	160
Hosts and Other Processors	165
So Many Ways of Communicating	166

OSI Model	166
Ethernet and WiFi	168
Serial	168
Serial Peripheral Bus Profiles	170
Parallel	176
Putting Peripherals and Communication Together	178
Data Handling	178
Adding Robustness to the Communication	188
Changing Data	191
Changing Algorithms	194
Further Reading	195
7. Updating Code	199
On-Board Bootloader	200
Build Your Own Bootloader	201
Modifying the Resident Bootloader	203
Brick Loader	204
Linker Scripts	205
Copy Loader to RAM	208
Run the Loader	209
Copy New Code to Scratch	210
Dangerous Time: Erase and Program	210
Reset to New Code	211
Security	212
Summary	212
8. Doing More with Less	215
Code Space	216
Reading a Map File (Part 1)	216
Process of Elimination	219
Libraries	220
Functions and Macros	221
Constants and Strings	222
RAM	223
Free malloc	223
Reading a Map File (Part 2)	228
Registers and Local Variables	229
Function Chains	231
Pros and Cons of Globals	232
Memory Overlays	233
Speed	234
Profiling	235
Optimizing	239

Summary	249
Further Reading	250
9. Math	253
Identifying Fast and Slow Operations	254
Taking an Average	255
Use an Existing Algorithm	258
Designing and Modifying Algorithms	260
Factor Polynomials	261
Taylor Series	261
Dividing by a Constant	264
Scaling the Input	265
Lookup Tables	266
Fake Floating Point Numbers	274
Precision	275
Addition (and Subtraction)	276
Multiplication (and Division)	277
Determining the Error	278
Further Reading	282
10. Reducing Power Consumption	285
Understanding Power Consumption	286
Turn Off the Light When You Leave the Room	288
Turn Off Peripherals	288
Turn Off Unused I/O devices	289
Turn Off Processor Subsystems	289
Slowing Down to Conserve Energy	290
Putting the Processor to Sleep	290
Interrupt-based code flow model	291
A Closer Look at the Main Loop	294
Processor Watchdog	295
Avoid Frequent Wake-ups	296
Chained Processors	296
Further Reading	296

Preface

I love embedded systems. The first time a motor turned because I told it to, I was hooked. I quickly moved away from pure software and into a field where I can touch the world. Just as I was leaving software, the seminal work was done on design patterns.*My team went through the book, discussing the patterns and where we'd consider using them. As I got more into embedded systems, I found compilers that couldn't handle C++ inheritance, processors with absurdly small amounts of memory in which to implement the patterns, and a whole new set of problems where design patterns didn't seem applicable. But I never forgot the idea that there are patterns to the way we do engineering. By learning to recognize the patterns, we can use the robust solutions over and over. So much of this book looks at standard patterns and offers some new ones for embedded system development. I've also filled in a number of chapters with other useful information not found in most books.

About This Book

After seeing embedded systems in medical devices, race cars, airplanes, children's toys, and gunshot location systems, I've found a lot of commonalities. There are a lot of things I wish I knew then on how to go about designing and implementing software for an embedded system. This book contains some of what I've learned. It is a book about good software design in resource constrained environments.

It is also a book about understanding what interviewers look for when you apply for an embedded systems job. Each section ends with an interview question. These are generally not language specific; instead they attempt to divine how you think. Good interview questions don't have a single correct answer. Instead of trying to document all the paths, the notes after each question provide hints about what an interviewer might look for in your response. You'll have to get the job (and the answers) on your own merits.

* Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software* .

One note, though: my embedded systems don't have operating systems. The software runs on the bare metal. When the software says “turn that light on,” it says it to the processor without an intermediary. This isn't a book about an embedded operating system (OS). But the concepts translate to processors running OSs, so if you stick around, you may learn about the undersides of OSs too. Working without one helps you really appreciate what an OS does.

This book describes the archetypes and principles that are commonly used in creating embedded system software. I don't cover any particular platform, processor, compiler or language, because if you get a good foundation from this book, specifics can come later.

About the Author

In the field of embedded systems, I have worked on DNA scanners, inertial measurement units for airplanes and race cars, toys for preschoolers, a gunshot location system for catching criminals, and assorted medical and consumer devices.

I have specialized in signal processing, hardware integration, complex system design, and performance. Having been through FAA and FDA certification processes, I understand the importance of producing high-quality designs and how they lead to high-quality implementations.

I've spent several years in management roles, but I enjoy hands-on engineering and the thrill of delivering excellent products. I'm happy to say that leaving management has not decreased my opportunities to provide leadership and mentoring.

Organization of the Book

I read nonfiction for amusement. I read a lot more fiction than nonfiction but, still, I like any good book. I wrote this book to be read, almost as a story, from cover to cover. The information is technical (extremely so in spots) but the presentation is casual. You don't need to program along with it to get the material (though trying out the examples and applying the recommendations to your code will give you a deeper understanding).

This isn't intended to be a technical manual where you can skip into the middle and read only what you want. I mean, you can do that, but you'll miss a lot of information with the search and destroy method. You'll also miss the jokes, which is what I really would feel bad about. I hope that you go through the book in order. Then, when you are a hip deep in alligators and need to implement a function fast, pick up the book, flip to the right chapter and, like a wizard, whip up a command table or fixed point implementation of variance.

Or you can skip around, reading about solutions to your crisis of the week. I understand. Sometimes you just have to solve the problem. If that is the case, I hope you find the chapter interesting enough to come back when you are done fighting this fire.

The order of chapters is:

Chapter 1. Introduction

What is an embedded system? How is development different from traditional software?

Chapter 2. Creating a System Architecture

How to create (and document) a system architecture.

Chapter 3. Getting the Code Working

Hardware/software integration during board bring up can be scary, but there are some ways to make it smoother.

Chapter 4. Outputs, Inputs and Timers

The embedded systems version of “Hello World” is making an LED blink. It can be more complex than you might expect.

Chapter 5. Task Management

This chapter describes how to set up your system, where to use interrupts (and how not to), and how to make a state machine.

Chapter 6. Communicating with Peripherals

Different serial communication forms rule embedded systems (UART, SSP, SPI, I2C, USB, etc.). Networking, bit-bang, and parallel buses are not to be discounted.

Chapter 7. Updating Code

When you need to replace the program running in your processor, you have a few options from internal bootloaders to building your own solution.

Chapter 8. Doing More with Less

This covers methods for reducing consumption of RAM, code space, and processor cycles.

Chapter 9. Math

Most embedded systems need to do some form of analysis. Understanding how mathematical operations and floating point work (and don't work) will make your system faster.

Chapter 10. Reducing Power Consumption

From reducing processor cycles to system architecture suggestions, this chapter will help you if your system runs on batteries.

The information is presented in the order that I want my engineers to start thinking about these things. It may seem odd that architecture is first, though most people don't get to it until later in their careers. However, I want the people I work with to be thinking about how their code fits in the system long before I want them to worry about optimization.

Conventions Used in This Book

Typography

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, data types, and keywords.

Constant width bold

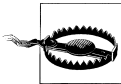
Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Terminology

A microcontroller is a processor with on board goodies like RAM, code space (usually flash) and various peripheral interfaces (e.g. I/O lines). You code runs on a processor or central processing unit (CPU). A microprocessor is a small processor. But the definition of small changes.

A DSP (digital signal processor) is a specialized form of microcontroller which focuses on signal processing, usually sampling analog signals and doing something interesting with the result. Usually a DSP is also a microcontroller but it has special tweaks to make it perform math operations faster (in particular multiply and add).

As I wrote the book, I wanted to put in the right terminology so you'd get used to it. However, this is so critical I don't want to keep changing the name. Throughout the book, I stick with the term processor to represent whatever it is you are using to implement your system. Most of the material is applicable to whatever you actually have.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Making Embedded Systems* by Elecia White (O'Reilly). Copyright 2011 Some Copyright Holder, 9781449302146.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at <http://my.safaribooksonline.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

[*http://www.oreilly.com/catalog/9781449302146*](http://www.oreilly.com/catalog/9781449302146)

To comment or ask technical questions about this book, send email to:

[*bookquestions@oreilly.com*](mailto:bookquestions@oreilly.com)

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

[*http://www.oreilly.com*](http://www.oreilly.com)

Introduction

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 non-technical 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 running the application. The hardware often has constraints: for instance, a CPU that runs more slowly to save battery power; a system that uses less memory so it can be manufactured more cheaply; 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 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 where servicing faulty software or broken hardware may be infeasible (i.e. 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 (a heart monitor should not fail quietly).

Compilers, Languages, and Object-Oriented Programming

Another way to identify embedded systems is that they use cross compilers. While 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. Since 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, many embedded C++ compilers implement only a subset of the language (commonly, multiple inheritance, exceptions, and templates are missing). There is a growing popularity for Java, but the memory management inherent to the language works only on a larger system.

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 (e.g. a remote control talking to a set top box, 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 as I've said, good design principles transcend language.

Embedded System Development

Embedded systems are special, offering special 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.

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"), whether it actually implements that widespread standard or not.

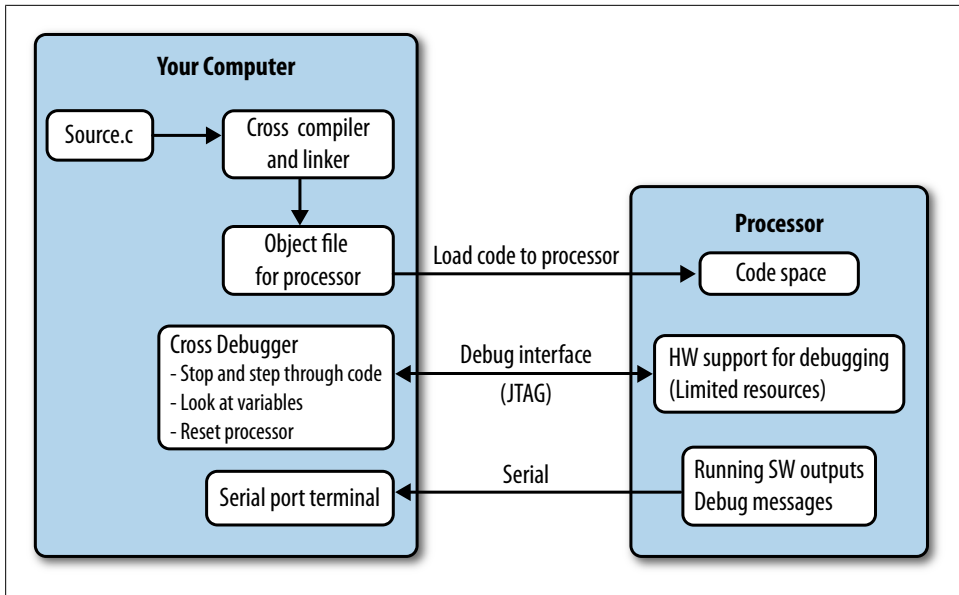


Figure 1-1. Computer and target processor

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 machine code to say “stop here.” However, if your code is programmed to flash (or any other sort of read-only memory), instead of modifying the machine 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 there are often only a limited number of hardware breakpoints available.

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 an emulator, an in-circuit emulator (ICE), or a JTAG adapter. These may refer (some-

what incorrectly) to the same thing or they may be three different devices. The emulator is specific to the processor (or processor family), so you can't take the emulator you got for one project and assume it will work on another. The emulator costs add up, particularly if you collect enough of them or if you have a large team working on your system.

To avoid buying an emulator or dealing with the processor limitations, many embedded systems are designed to have their debugging primarily done via `printf` or some sort of lighter weight logging to an otherwise unused communication port. While 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.

More Challenges

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)
- Processor cycles or speed
- Battery life (or power savings)
- Processor peripherals

To some extent, these are interchangeable. 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 obvious 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 be challenged to learn quickly.

Once development and testing are finished, the system is manufactured, 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 a challenge 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, the needs of the project 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 may not support change as the application morphs. Engineering embedded systems is not just about strict constraints and the eventual life of the system. The challenge 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 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 which shows how it 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 challenges 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 have an impact on 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 difficult 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 old-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.

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

You've got only a limited amount of time: focus on where you can get better results by looking for the bigger resource consumers after you've got 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.

Further Reading

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

1. Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2. 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 at the “Gang of Four” book (or a standard design pattern may be noted as a GoF pattern).
2. Freeman, Eric T; Elisabeth Robson, Bert Bates, Kathy Sierra (2004). Head First Design Patterns. O'Reilly Media. ISBN 0-596-00712-4.

Interview question: hello world

Here is a computer with compiler and an editor. Please implement “hello world”. Once you've got 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 (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 to be able to 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).



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 func-

tion, 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 is 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 who asks this question gives bonus points if a candidate can, on 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.

Creating a System Architecture

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.

Starting a project from scratch can be difficult. 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 white board, 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 that the hardware will be based on an existing platform with some changes.

Embedded systems depend heavily on their hardware. Unstable hardware leaves the software looking buggy and unreliable. In this section, we'll look at creating a system architecture from the general hardware level and going up to the function level. 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 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 bottoms-up design as described here, recognize that the hardware you are specifying is archetypal. While 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.

Creating System Diagrams

Just as hardware designers create schematics, you should make a series of software block diagrams that show the relationships between the various parts of the software system. 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 three different diagrams:

- Architecture block diagram
- Hierarchy of control organization chart
- Software layering view

The Block Diagram

Design is straightforward at the start because you are considering the physical components of the system, and you can think in an object-oriented fashion—whether or not you are using an object-oriented language—to model your software around the physical components. 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 a chip is in the center, the communication objects are in the processor and the peripherals are each attached to those.

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 later). Most processors cannot execute code over SPI so the flash is used for off-processor storage. Our schematic, shown at the top of [Figure 2-1](#), shows that we have some flash memory attached to our processor via SPI.

In our software block diagram, we'll add the flash as a peripheral (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 peripheral; 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 2-1](#) 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, particularly where the details may have a system impact, but try to keep the details out of the diagram if possible.

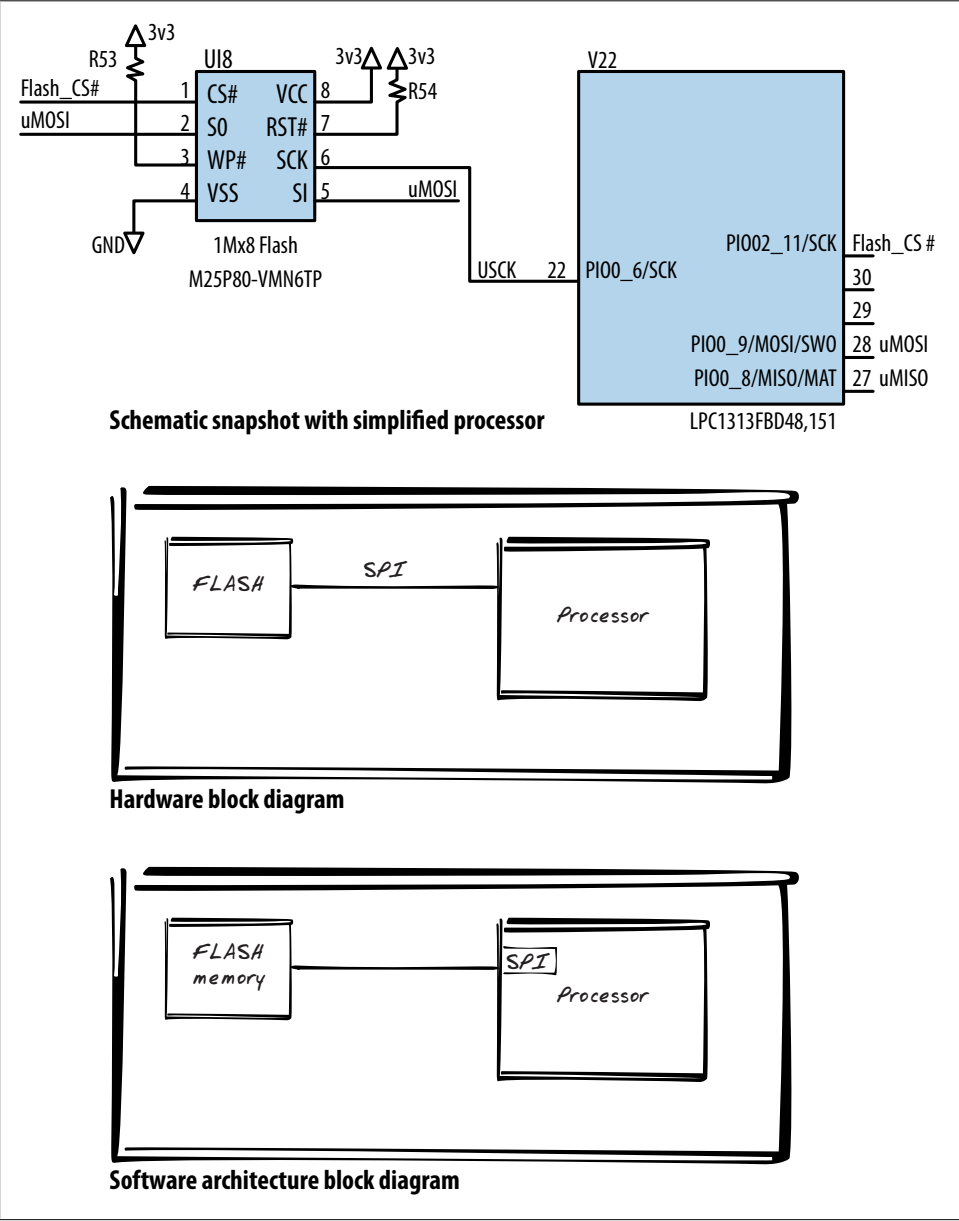


Figure 2-1. Comparison of schematic and initial software block diagram

The next stage is to add some higher level functionality. What is each peripheral chip used for? This is simple if 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 component, 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 machine, etc. You may not know exactly what you'll need for those (some of them we'll talk about more later in the book) but try to represent everything from the hardware to the product function in the diagram. See [Figure 2-2](#).

After you have sketched out this diagram on a piece of paper or a white board (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, another drawing may give additional insight.

Looking at the various views may show you some hidden, ugly spots with critical bottlenecks, poorly understood requirements, or an innate failure to implement the product on the platform. Often these deficiencies can be seen from only one angle or are represented by the boxes that most change between the different diagrams. By looking at them from the right perspective, you may not only identify the tricky modules but also see a path to making a good solution.

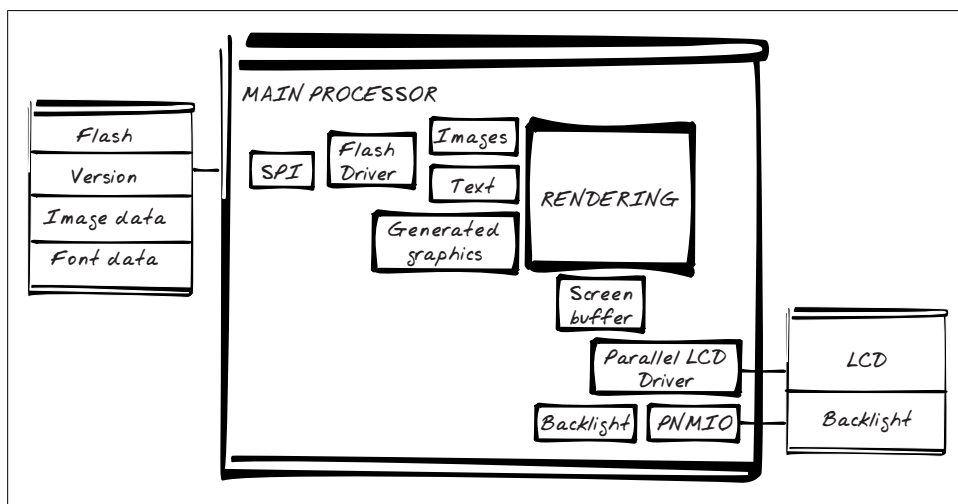


Figure 2-2. Software block diagram

Hierarchy of Control

The next type of software architecture diagram looks like an organizational chart, as in [Figure 2-3](#). Main is the highest level. If you 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. We can then fill in the lower-level objects that are used by a higher-level object. So 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).

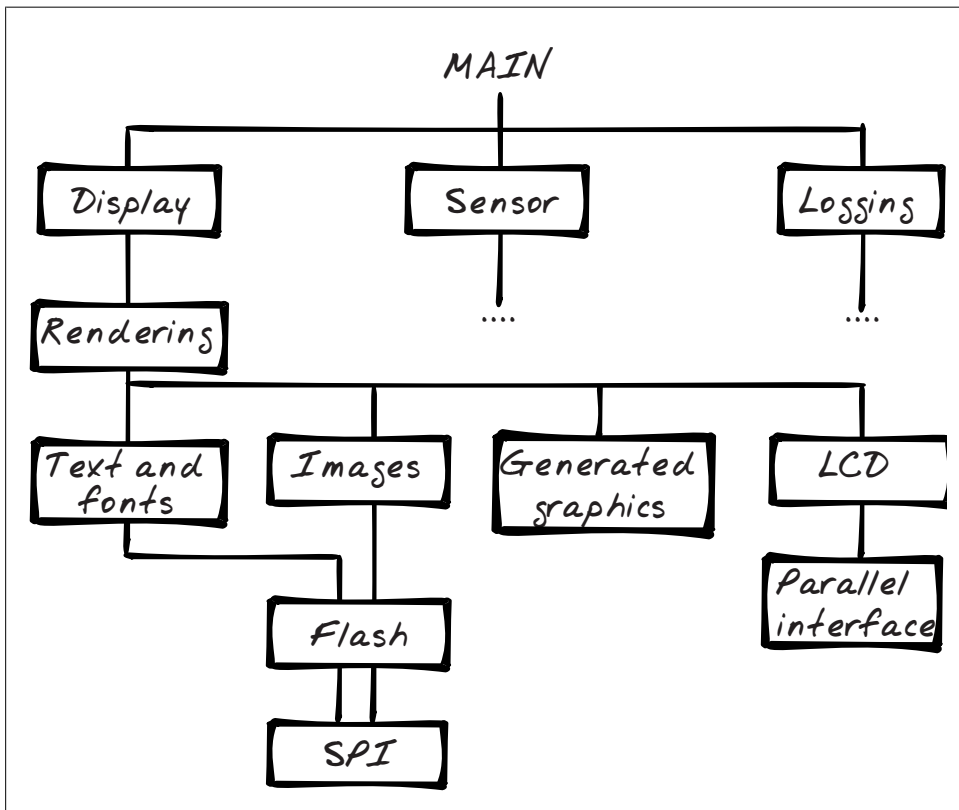


Figure 2-3. Organizational diagram

However, as much as we'd like to devote a peripheral 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 peripheral.

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 there is unlikely to be a conflict between them.

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 peripheral, 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.

In [Figure 2-4](#), 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've got two threads or my display assets are used in an interrupt), the software will have to avoid collisions and any resulting corruption.

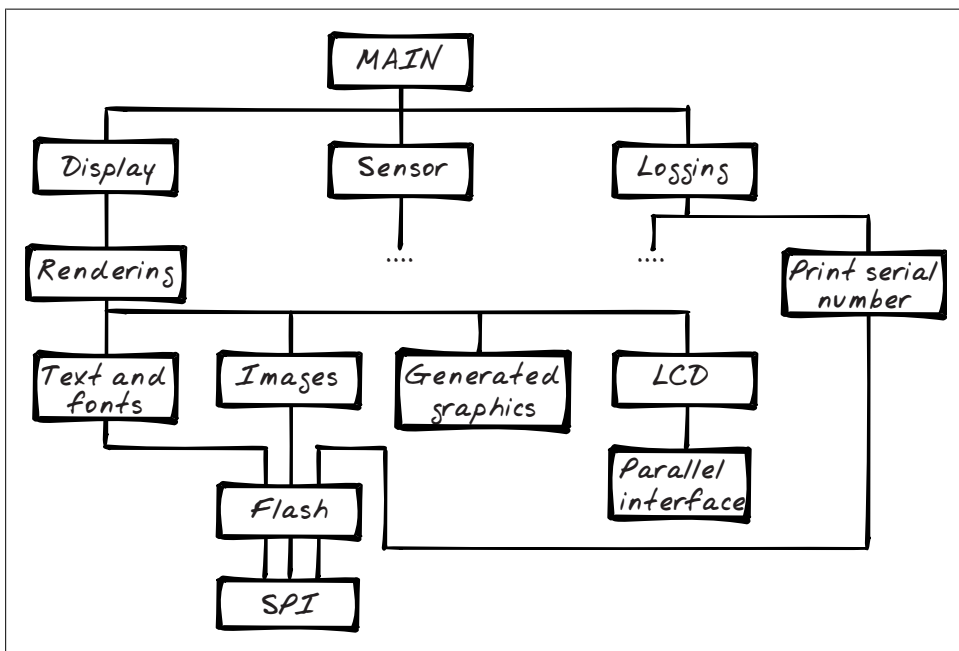


Figure 2-4. Organizational diagram with a shared resource

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 maintenances 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.

Layered View

The last architecture drawing looks for layers and represents objects by their estimated size, as in [Figure 2-5](#). 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 (like 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 the items that use the lowest layer to the diagram. If there are multiple users of a lower level object, they should all touch the lower level object (this might mean making an object 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 so many things they can't all touch, it will probably increase in complexity even if the goal of the module is 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, ultimately, on the project that I took this snippet from, rendering became a relatively large module and then two modules.

Eventually, the layered view 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. Possibly 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 only dealt 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 maintainability.

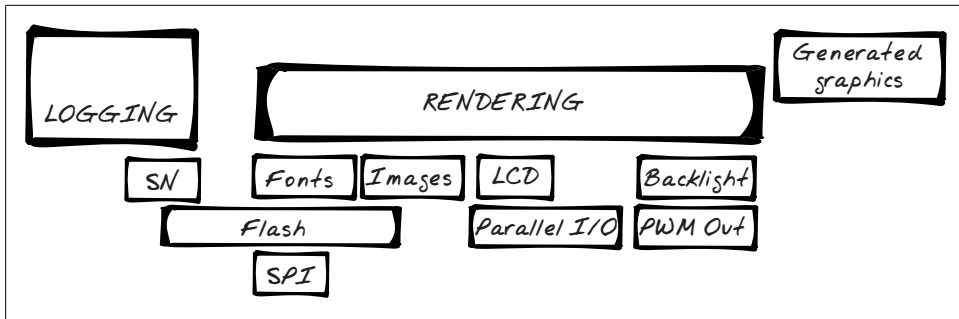


Figure 2-5. Layered software diagram

From Diagram to Architecture

So now, sitting with three 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.

Given your product requirements, you may be pretty confident about some of the functions of your system. 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 may change. The LCD, the image, or font data may also change. Even the way you store the image or font data may change. The boxes in the diagram should represent the Platonic ideals of each thing instead of a specific implementation.

Encapsulate Modules

Thus we are making interfaces between the modules that don't depend specifically on what is in them (this is encapsulation!). We use the three 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 flexibility in the future, it may be worth collapsing some dependency trees. Here are some things to look for:

- In the hierarchy chart, look for objects that are used only by one other object. Are these both fixed? Or can one change independently of the other?

- In the layered 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?

In whole, as you look at the architecture drawings, consider each box as a software file, a module, or possibly an object. What interfaces does it have to its neighbors?

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?



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.

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

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).

Many drivers in embedded systems are based on API used call devices in Unix systems. Why? Because the model works well in many situations and it 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`

(Pronunciation: 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 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:*

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

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

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 someone comes up to your code, if the driver looks like it has these functions, they will know what to expect.



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 times, 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 Figure 2-6](#). 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.

Note that drivers are stackable, as shown in [Figure 2-7](#). In our example, we've got a display that uses flash memory that 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.

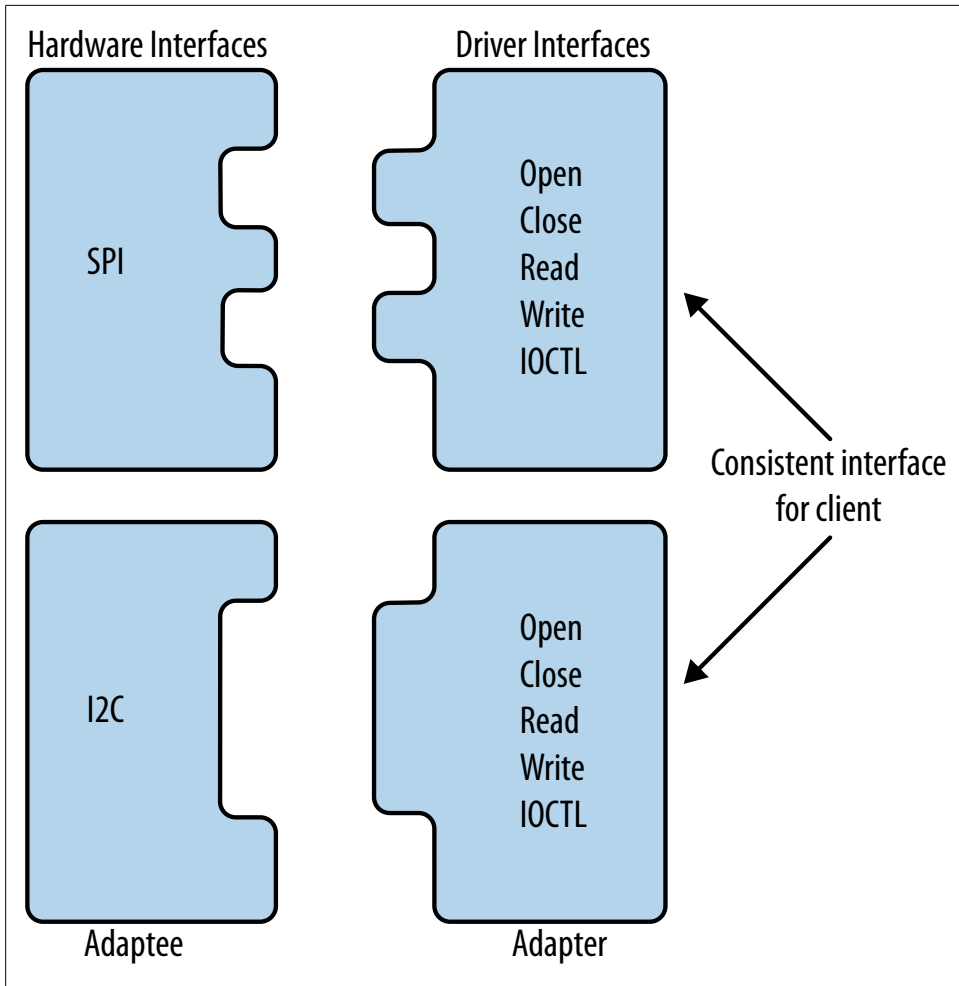


Figure 2-6. Drivers implement the Adapter pattern

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 may not need to change, or may only need to replace flash functions with EEPROM ones.

In [Figure 2-7](#), 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 place holders.

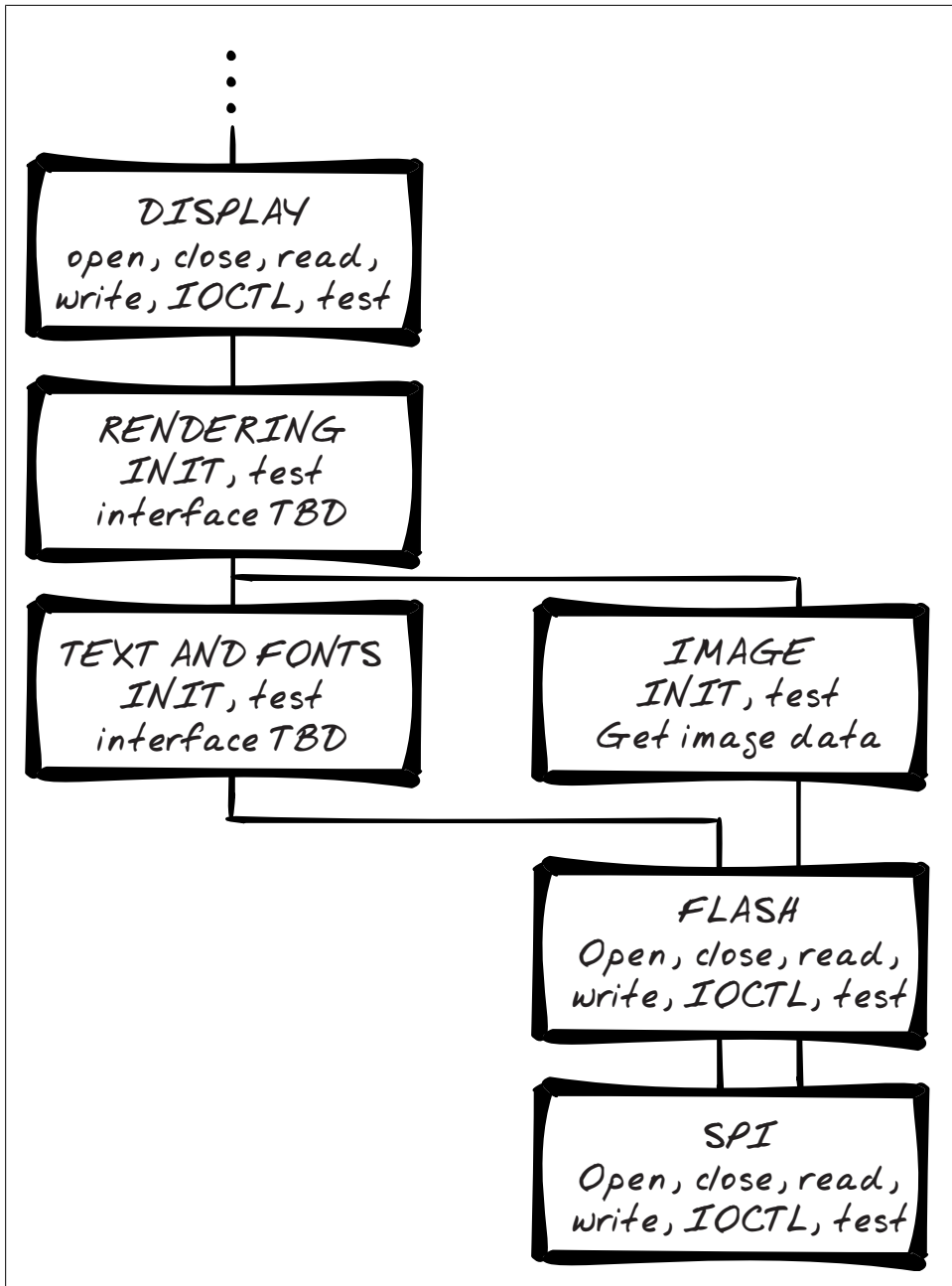


Figure 2-7. Interfaces for display subsystem and below

Getting Started With Other 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, start working out the responsibilities of each box in your architecture diagrams.



Having created three versions of the architecture, you may not 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

A resource constrained system often lacks a path to let your code talk to the outside world. The goal of the logging module noted in the example system is to implement a robust and usable 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.



Logging debug output can slow down a processor significantly. If your code behaviour 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 debug-able is part of making it maintainable. Even if your code is perfect, the person who comes after you may 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.

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 you 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 RS232 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, but not on the custom hardware. So the first big requirement for the module is: 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 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.

Typical calls needed for logging

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 and not a variable argument like `printf` or `iostream`. 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 needing 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:

```
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. While these may 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 (UART, I2C, other bus, etc.). 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.

The 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 may not be available so you may pare your output back to an LED or two.

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 to every function that has a function that needs to log something.



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.

Maybe this isn't a good idea. How can every user of the logging subsystem get access to it? As noted in [“Object Oriented Programming in C” on page 26](#), you can create some object oriented characteristics in C. 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;
```



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 else 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;
    static 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 = 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 have 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* Instance() {
        if (mInstance == 0) {
            mInstance = new Singleton;
        }
        return mInstance;
    }
protected:
    Singleton(); // no one can create this except itself
private:
    static Singleton* mInstance = 0;
}

```

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 which 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” on page 26](#). 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've still got 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 independently developed and tested.

In this pattern, the view is the interface to the user, both input and output. In our device, the user may not be a person: it may 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. It 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 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 Model-View-Controller 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 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 a picture of a sun or a detailed read-out 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 they can each be reused. For that MP3 player, your company may want a consistent interface even though your audio playing hardware was 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, 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 action on the model.

Figure 2-8 shows a basic interpretation of the model view controller 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 murderous or cheerfully optimistic. You may need some context clues to know which it is, but you are probably talking about someone's mood either way.

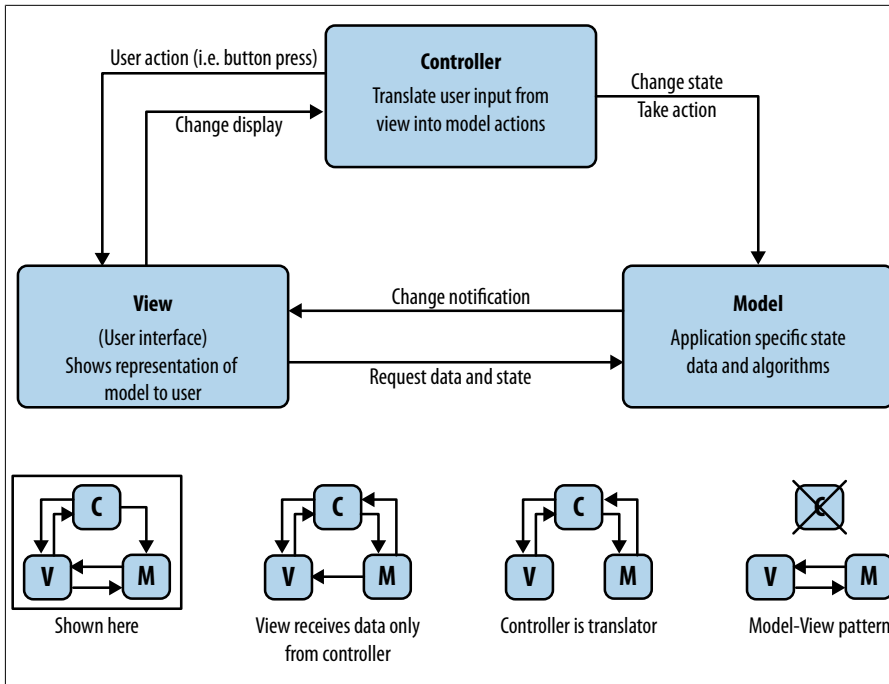


Figure 2-8. Model View Controller Overview

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.

Take all of the inputs to the algorithm's box and make an interface that lets you quickly and easily generate them from your host system or from a file on a PC. 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 little 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 shouldn't change. The controller is the part that changes when you put the algorithm on a PC. Consider an MP3 player (Figure 2-9) and what the MVC diagram might look like with a sandbox. Note that the view and controller will both have relatively strict APIs because you will be replacing those as you move from virtual device to actual device.

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           // should do all of the initialization without further interaction
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 input 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, 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 file name and the resulting bitstream of information that would go to the DAC could be considered the view (which consists of inputs and outputs, whether or not they go to humans), 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 might 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 could potentially 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 debug-ability 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.

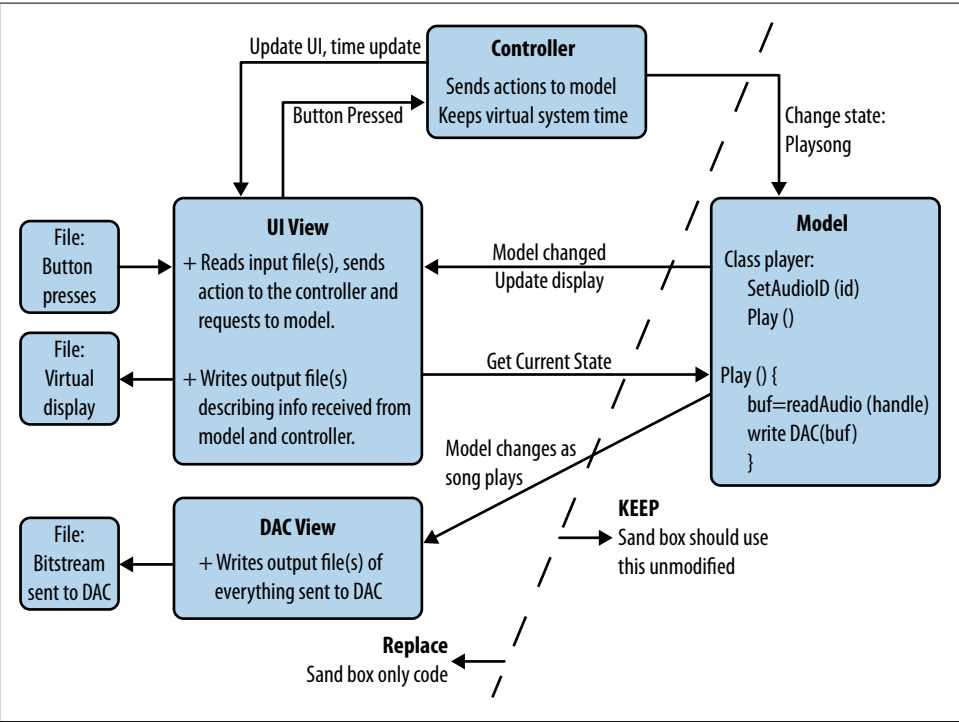


Figure 2-9. Model-View-Controller in a sandbox

Proceeding in the Face of Resource and Time Constraints

Creating an architecture diagram from scratch can be daunting. It can be even harder if you join a team with a close deadline, a bunch of code, and very little documentation with no time to write more. A system architecture diagram is still called for, though you may need to keep it in your notebook and fill it in as best you can. Your ability to see the 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.

The schematic (or the hardware block diagram if they've got it) is still a good place to start. After that, 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, one that may take a while to figure out.

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 the architecture gets bigger, it may need to have some depth to it, particularly for a mature design. If the drawing is too complex, bundle up some boxes into one box and then expand them on another drawing.

Which style of drawing you use for this depends on what you feel comfortable with. You may need to try a few different options before settling on one.

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*. This is the original, seminal work on design patterns. It uses C++ as the reference language.
- Freeman, Elisabeth; Eric Freeman, Bert Bates, and Kathy Sierra (2004), *Head First Design Patterns*. Using Java as the example language, this book gives great examples with an engaging style.
- Search on Wikipedia for *software design pattern*.

Interview question: Create an architecture

Describe the architecture for (pick an object in the room) this 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 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 they mentally de-construct 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. Candidates shouldn't be afraid to pick up the object and see the connections it has. Those are input and outputs as well. Once they've got the physical hardware down, they can start making connections by asking (themselves) 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 (i.e. is there an analog to digital converter)?

Candidates get 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 them to ask questions about specific features or possible design goals (cost). However, they get a lot of latitude to make up the details. Want to talk about net-

working? 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 they talk about things they are interested in, particularly areas that I've failed to ask them about in other parts of the interview.

While 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 their experience would help them design a phone. If I ask a question, I don't mind if they admit their ignorance and talk about something they do know.

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 seeing the candidate show enthusiasm in problem solving and effectively communicate his ideas.

Getting the Code Working

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 good 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. 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
- Having a 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](#)).

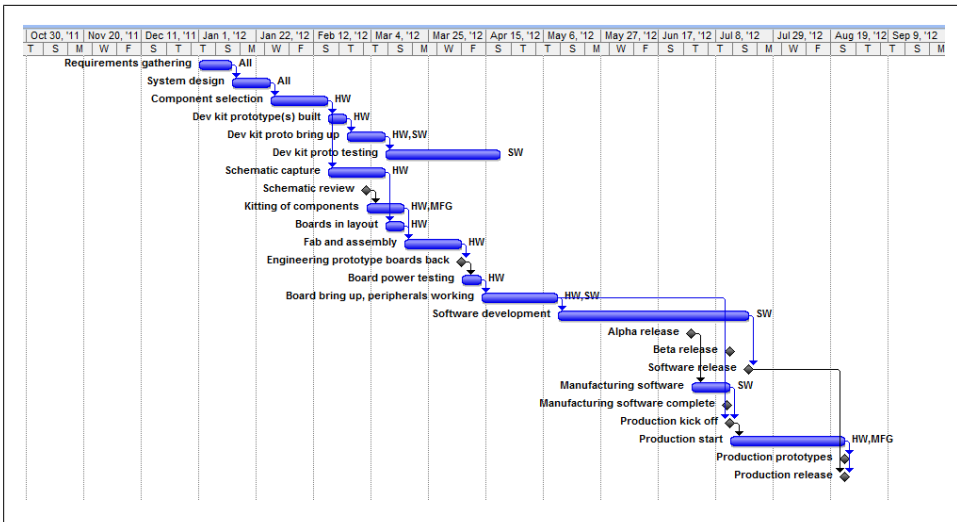


Figure 3-1. Ideal Project Schedule

Ideal Project Flow

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 kit. 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, etc. 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 (see [Figure 3-1](#)).

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 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” on page 51](#)), 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 map of physical traces on a board, connecting each component as they are in the schematic.



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. A board and all of its loose components are *kits*. Getting the long lead time parts can make it difficult to create a complete kit, which can delay assembly. An assembled board is called a PCBA, which is what will sit on your desk (or in your lab).

After the schematic is done and layout started, the primary task for the embedded software team is to define the hardware tests and get them written while the boards being are created. The algorithms are more fun and creating an outer shell for the system looks 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, they 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/or try them out on a development kit).

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 (finally!) you get a board and bring up starts.

Board Bring Up

This can be a fun time, where you are learning from an engineer whose skill set is different than 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 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; your long-term outlook will be good.



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 project 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 make sure the motor moves at all.

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; you'll probably want it again when the next set of boards come 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 knowing an in-depth knowledge of the processor and peripherals. That leads to the topic of reading a datasheet.

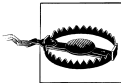
Reading a Datasheet

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. All the effort you'd need to put into learning a software package (Qt, Gtk, Boost, STL, etc.),

you are going to need to put into learning your chip and peripherals. They will even have methods of talking to them 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. So make sure you've got 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.



Unfortunately, many manufacturers release their datasheets only under NDA so their website is 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. Near the top of each datasheet is an overview or feature list. While this is the summary of the chip, 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 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 that buy components in volume; it is the experienced engineers who are already familiar with the devices.

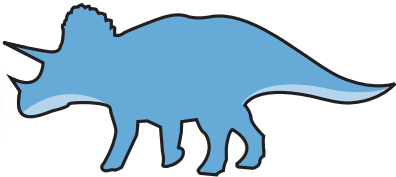
<div> <div>Dino Industries</div> <div>  </div> <div> T. HORRIDUS T. PROPRSUS </div> </div>	
Analog Triceratops	
Features	Description
<ul style="list-style-type: none"> - Ceratopsid genus - Three hooved hands - No fenestrae frills - Ultrasmall brain - Supply range: 2.7V to 5V 	Triceratops is a genus of herbivorous ceratopsid dinosaur which lived during the late Maastrichtian stage of the Late Cretaceous Period, around 68 to 65 million years ago in what is now North America.

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 or underline the important information (which could be most of the information).

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. Someday, when you aren't under a lot of pressure to get the product done, it is worth going through those sections.

You probably still want to know about these sections. These are the ones that are needed 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 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 slashes 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.

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 if 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 the first reading. However, when your component communicates but doesn't work, the performance characteristics can help you determine if there might be a reason (i.e. the part is rated to work over 0-70C 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 if 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 that 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 may be important.

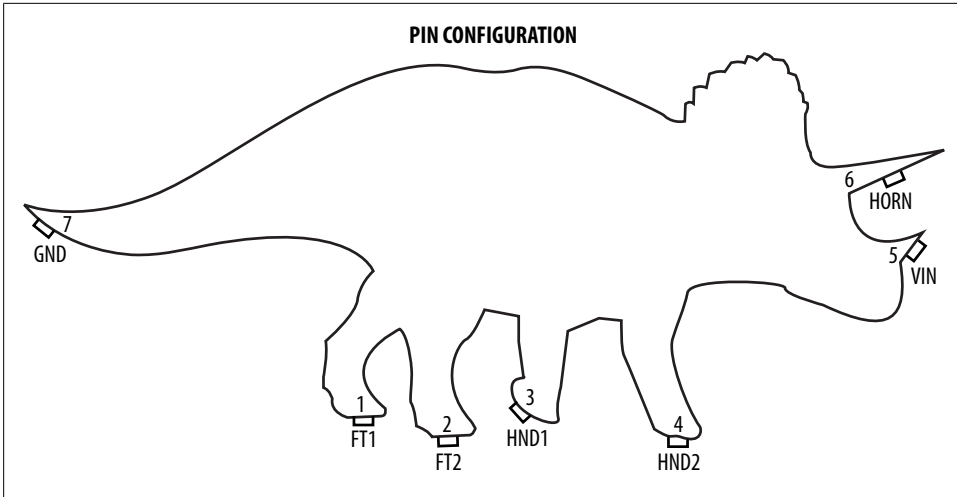


Figure 3-3. Analog Triceratops pin out

PIN DESCRIPTIONS			
SYMBOL NAME	PIN	Type	DESCRIPTION
/FT1	1	O	Four hoofed foot channel 1
/FT2	2	O	Four hoofed feet channel 2
/HND1	3	O	Three hoofed hand channel 1
/HND2	4	O	Three hoofed hand channel 1
VIN	5	I	Power supply
HORN	6	I	Mode setting
GND	7	I	Ground

Figure 3-4. Analog Triceratops pin descriptions

Important Text for Software Developers

Eventually, possibly halfway through the datasheet, you will get to some text ([Figure 3-5](#)). This could be titled the Application Information or the 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 you will eventually really need to read it from start to end. As a bonus, there the text 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 [“How Timing Diagrams Help Software Developers”](#) on page 46.

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

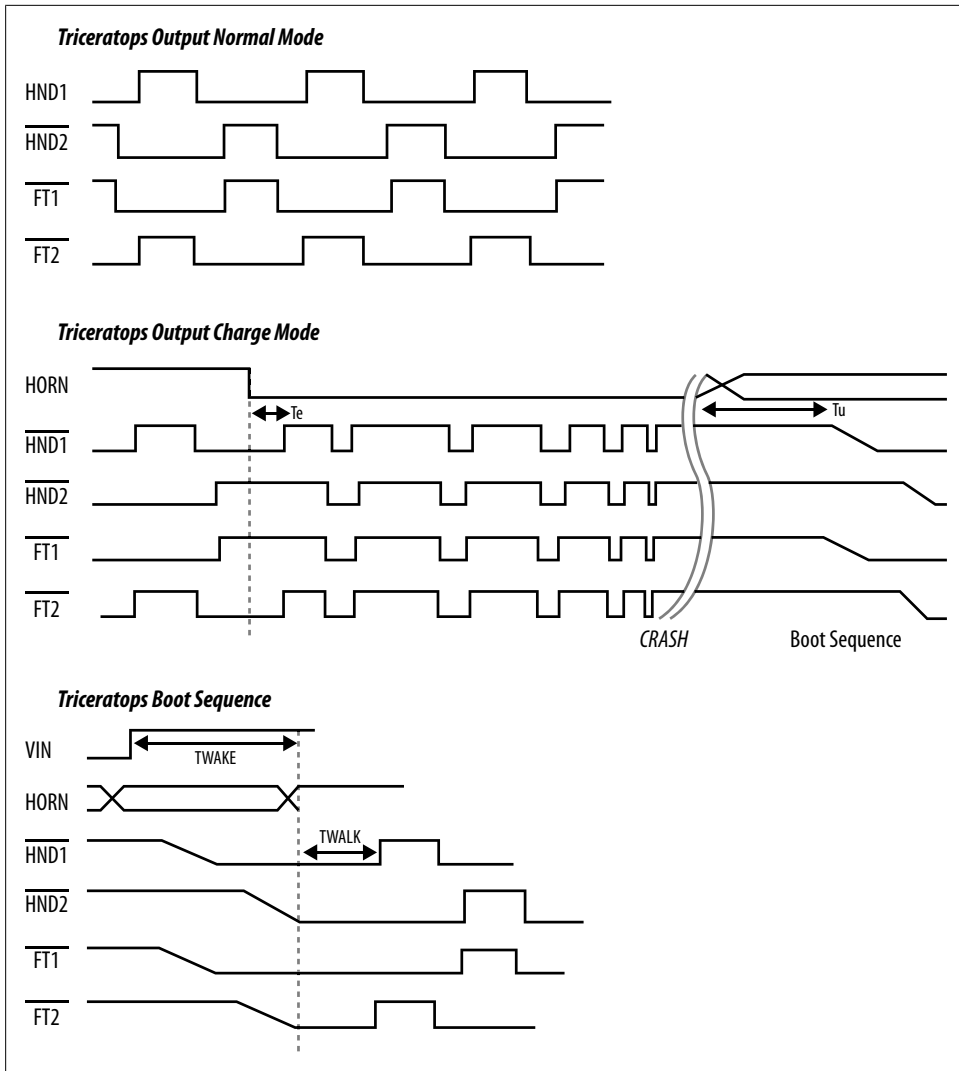


Figure 3-6. Analog Triceratops timing diagrams

Analog Triceratops Datasheet Errata

CLARIFICATIONS/CORRECTIONS TO THE DATASHEET

In the device datasheets listed below, the following clarifications and corrections should be noted. The component Triceratops may mature over time into the Torosaurus as light and temperature cycles cause the frills to develop fenestrae (holes). To work around the issue, the frill is not recommended for use in the Triceratops.

DEVICE	DATASHEET
<i>T. horridus</i>	Analog Triceratops
<i>T. prorsus</i>	
<i>T. latus</i>	Analog Torosaurus
<i>T. utahensis</i>	

Figure 3-7. Analog Triceratops errata

In addition to making sure you have the latest datasheet for your component, check the manufacturer's web page for errata, which provide 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.



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 using 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.

- There may be forums and examples of other people using the part.

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 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 develop before software engineers.

When you are evaluating a component, your goal is to eliminate components that won't work as fast as possible. Don't waste valuable time determining precisely how a component does feature X if the component can't be used because it requires 120V AC and your system has 5V 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 further investigation, let's talk about the things that datasheets don't have. They don't usually 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 the perfect part, it may only be available if you wait six months). Unless you are ordering online, you'll need to talk to your vendor or distributor.

It 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.

Back to the datasheet, this time those skipped sections become the most critical. 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 what 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 datasheet that meet your high standards, you can recheck the closest ones to see if 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 your functional parameters: Does the component go fast enough? Does the output meet or exceed that required by your system? In a sensor or an ADC, 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 in 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 not more than four datasheets. 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 only using one of them, this will give you a jump start 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 or 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 may 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
Input 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

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 the new ones become easier and easier.

While 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 methods called *registers*, which I'll cover in more detail in another chapter ([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 goal is to learn only 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) 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.



Why get the user manual from the vendor? Take for example the NXP LPC1313 processor, which uses an ARM Cortex-M3 core. You don't want to read the ARM user manual if you are using the LPC1313; 88% of the information in ARM manual is extraneous, 10% will be in the LPC13xx 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 LPC1313, you'll need to get a LPC13xx 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 will have usually 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 the 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 you orient yourself 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. While 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. 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 may have and 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 electrical focused. 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 may not 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 commu-

nications, 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 data-sheet, 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.

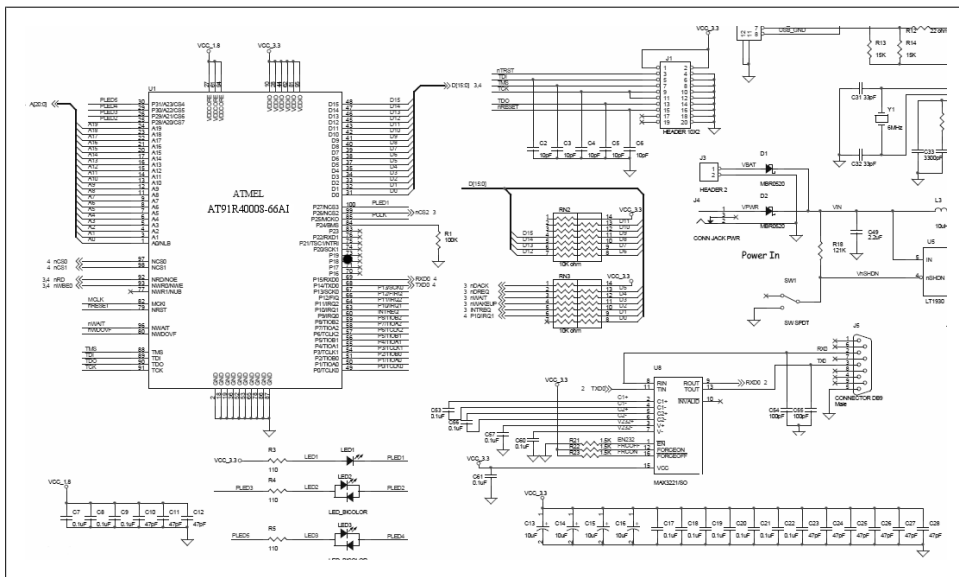


Figure 3-9. Example Schematic Snippet



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 number of wires— look for the largest boxes. Your processor is likely to be one of those things. Since it 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 may not be what you are used to seeing. While your processor manual may say Atmel AT91R40008, the schematic may have AT9140008-66AI which is more of a mouthful. However, the schematic not only describes how to make the traces on the printed circuit board, it also says 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 the 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 eight address lines and sixteen data lines as in Figure 3-9. Many newer processors have enough embedded memory to alleviate the need for externally memory mapped RAM or flash so you may not find any large components besides 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 RS232 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 like SENSOR or ADC might help. Chapter 6 chapter will give you some other signal names that might help you find interesting peripherals.



In a schematic, wires can cross without connecting. Look for a dot to indicate the connections.

In all this time, we've been ignoring the non-box shaped components. While 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 may be drawn a little differently in your schematic. This will let you express curiosity about “What does this resistor do?” “Further Reading” on page 71 gives you some suggestions on how to increase your hardware knowledge.

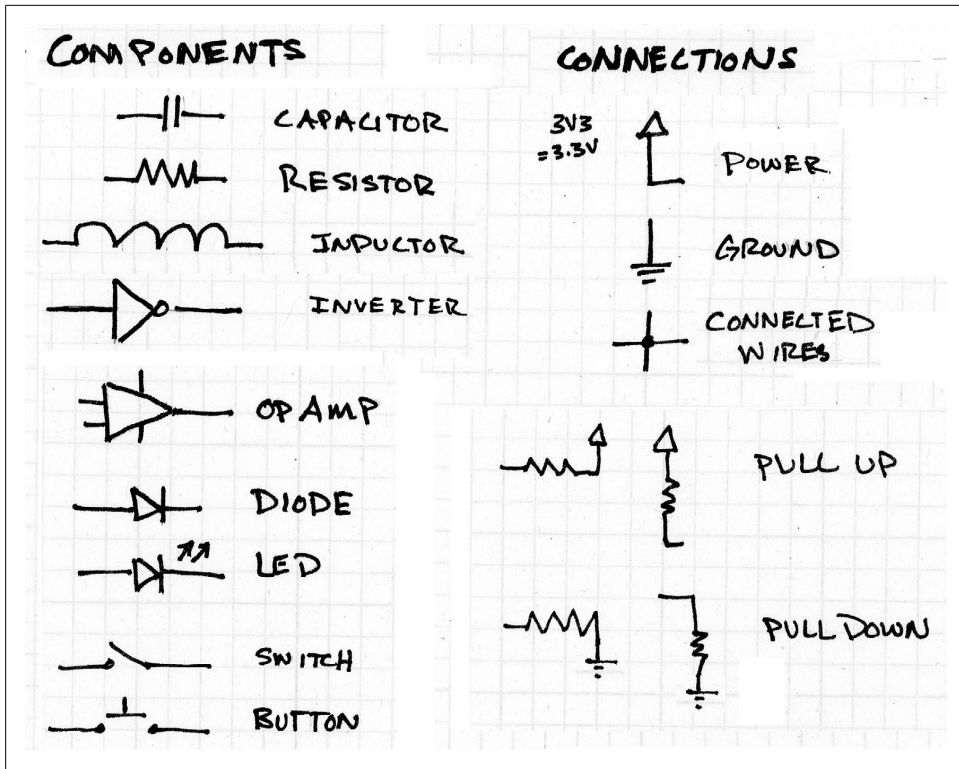


Figure 3-10. Common Schematic Components

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 it 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 (low 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 even 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 this applies to them except that their default logic level is low instead of high.

Having a Debugging Toolbox (and a Fire Extinguisher)

The datasheets, user manuals, and schematics we've examined so far 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 shock it.

Keep your Board Safe

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 having the hardware crushed under a falling book or being yanked off the table).

If possible, if any wires get added to the board, get them glued down as well as soldered. Many an hour has been lost to a broken reworked 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 mark 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 ask you how many boards you want allocated to 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. Not because you may very well damage the first one (or two).

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, RS232 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 their tools in labeled areas, use theirs. 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 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 0-20V with 0.1V granularity and from 0-2V with 0.01V granularity. The question you are usually looking to answer with the voltage mode of the DMM is simple: are any volts getting there at all? While a DMM with 1mV granularity might be nice occasionally, 80% of your DMM use will be the broader question of “Is the chip or component even powered?”

The second most important mode is the resistance check mode, usually indicated with the Ohm symbol 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 answers “Does it have 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; more on that 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 (high or low).

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 set up being complete. Now many of them hook to your computer and help you do the setup so that it 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 a 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.

Mixed signal oscilloscope

Combines the features of a traditional scope and logic analyzer with a couple 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 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 set is 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 probes to these signals (and attach the scope's ground clip to ground). 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 for the words to look for as you page through the manual.

Figure 3-11 shows what an archetypal oscilloscope screen looks like. 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.

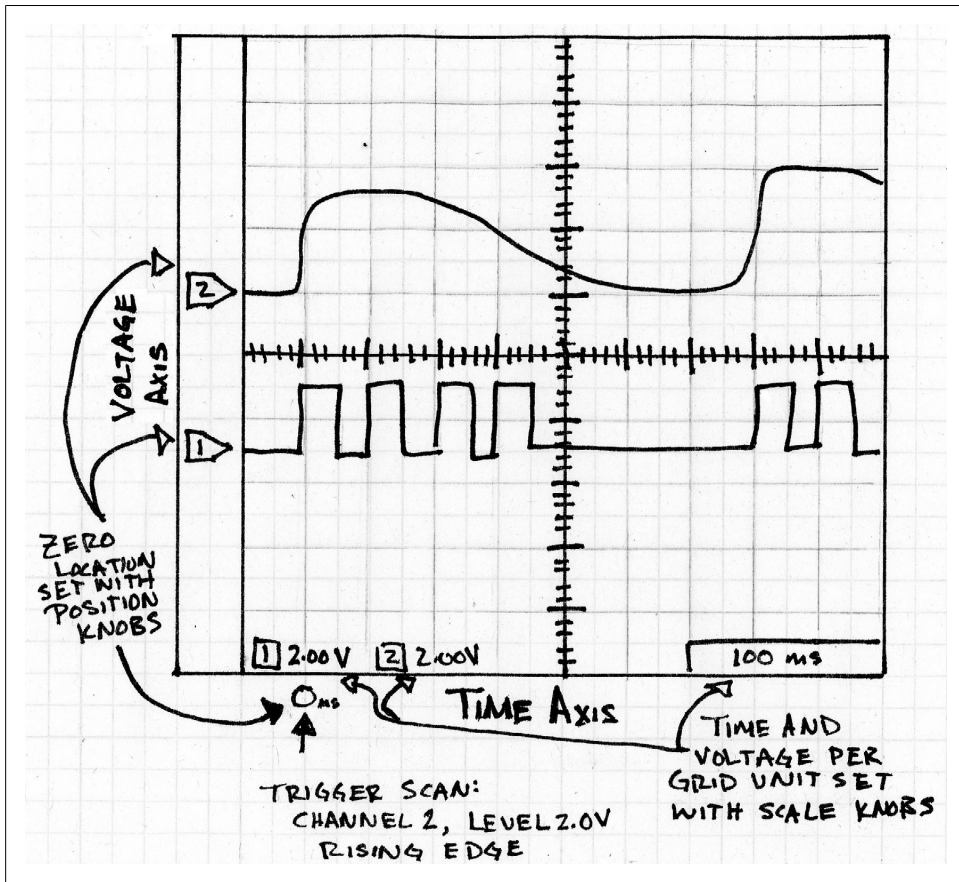


Figure 3-11. Oscilloscope Screen Example

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, 1s to 0.5s (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.



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

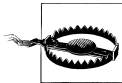
Once you set the time base, you'll need to set the scale of the voltage axis. This may be one knob that accesses all channels (or probes), making you shift the channel some other way, or you may get one knob per channel. Either way, as you turn it to the right, each vertical block magnifies (5V goes to 2V down to millivolts).

If you aren't sure, set the timescale to be about 100ms/block and the voltage granularity to be about 2V/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, 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.



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 automatically configure your scope for you. I find that it tends to reset the configuration to be 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, you usually 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.

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 being good with one requires a lot of practice. Don't get discouraged if it is a bit frustrating to start.

Testing the Hardware (and Software)

While I strongly recommend being ready to pull out the toolbox, DMM, and scope, 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 building the software that will test the hardware in a manner conducive to easy debugging.

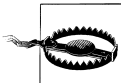
Three kinds of tests are commonly seen for embedded systems. First, the power on self-test (POST) runs every time you boot the system, even after the code is released. This test verifies that all of the hardware components are there to run your system safely. The more the POST tests, the longer the boot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



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 may 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.

The words *unit test* mean different things to different people. To me, it is automated test code that verifies that 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 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.



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](#)). The ones that aren't, I would encourage you to leave in the production code if you can, making them run 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, you may find that using the unit tests as a first line check in the field will tell you if 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 build systems. And once you've checked such code in to 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) is often 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 you may end up writing drivers for six peripherals before you get to integrate your software with the hardware, as you wait for the boards to get back.

In [Chapter 2](#), we had a system that communicated to a flash memory device via the SPI communication protocol (partial schematic reproduced in [Figure 3-12](#)). 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)

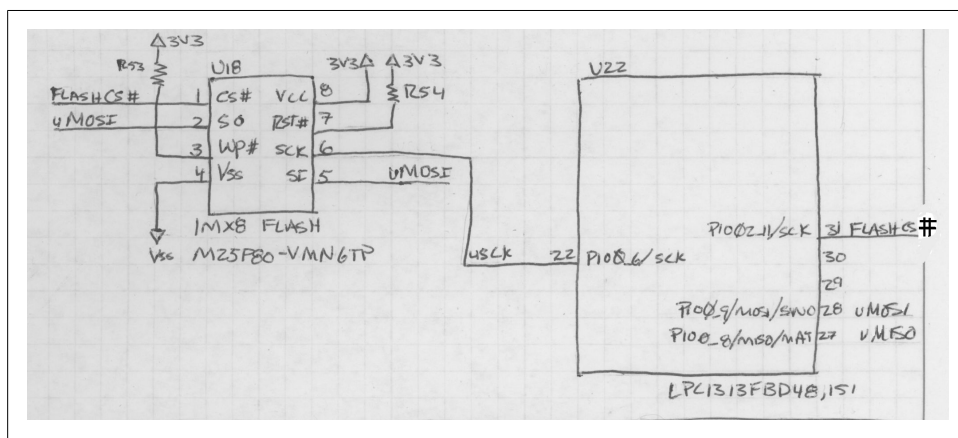


Figure 3-12. Flash schematic snippet

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 to have the others available for debugging as needed.

While other chapters cover controlling I/O lines ([Chapter 4](#)) and a bit more about SPI ([Chapter 6](#)), 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 Numonyx M25P80 flash memory datasheet (search for the part number on Google or Digikey).

Flash Test Example

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



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 tests component, a sector is 65536 bytes and the whole chip contains 1Mbyte (or 16 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 retain. 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. Because at least two of these tests have the potential to be destructive, we don't want to do them on power on (POST). Further, we don't need to. 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: bytes and multi-byte blocks. When running the code, you'll want to use the faster block access. As you test, though, the goal is to start out simple to build a good foundation.

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 an 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 just 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.



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, so 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:

1. Read the original data.
2. Write some changing but formulaic junk.
3. Verify the junk.
4. Re-write original.
5. Verify the original.

However, there are many other types of peripherals, more than I can cover here. While 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 just run 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 do not often have the rich user interface experience found in computers (or even smart phones). 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 3-13 shows some high level goals for our automated command handler. The figure specifies a port (serial or Ethernet) and a file read from external memory or any other communication method. We'll ignore those for now and concentrate on being able to send a command and get a response.

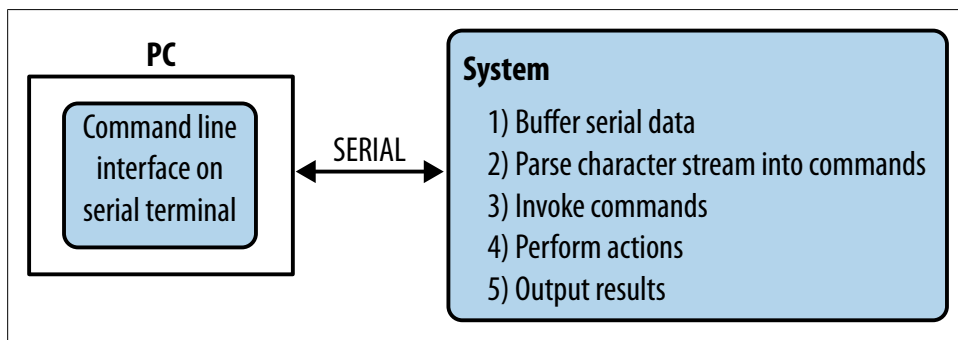


Figure 3-13. Goals for a command handler

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 more easily be able to extend it in unforeseen directions.

Creating a command

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

Ver

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 of the ways 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, [“Function pointers aren't so scary” on page 66](#) 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, printing out the number of errors upon completion"},
    {"blinkLed", &CmdBlinkLed, "Sets the LED to blink at a desired rate (parameter: frequency (Hz))"},
    {"",0,""} //End of table indicator. MUST BE LAST!!!
};
```

The command execution functions `CmdVersion`, `CmdFlashTest`, and `CmdBlink` 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, it 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 doesn't contain the help command. That is a special macro 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? Say for example 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 eFIRFilter:
        return fir(data, dataLen);
    case eIIRFilter:
        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:

```
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:

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

Once you are comfortable with the idea of function pointers, they can be a powerful asset when the code is supposed 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. Most processors try to predict where your code is going and load the appropriate in-

structions 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 are to 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 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.

Each command may perform the desired action itself or call another function (the receiver) to perform the action. For example, the peek command may get the address to be read and then just read the address. On the other hand, the blink LED command will probably call whatever function already exists to set interface to the LED and set a timer to make it blink. This function, the receiver, is the code that the user really wanted to talk to.

If a command implements the receiver (like the version command does), it is called a *smart command*. This is a misnomer, as separating the command and the receiver is usually smarter because it leads to a more easily extensible system.

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 3-14](#), there are four pieces:

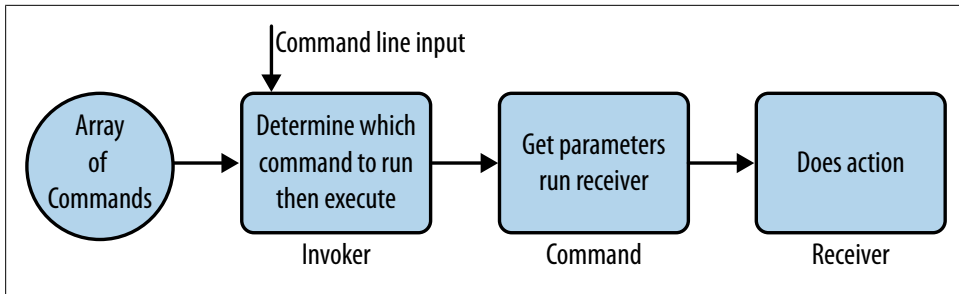


Figure 3-14. How the command handler work

Client

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 it was done in our example by creating an array) or create the associations on the fly.

Invoker

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 object

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

Knows how to service the request. This is goal code to be run.

Figure 3-15 shows the purpose of each element of the command pattern.

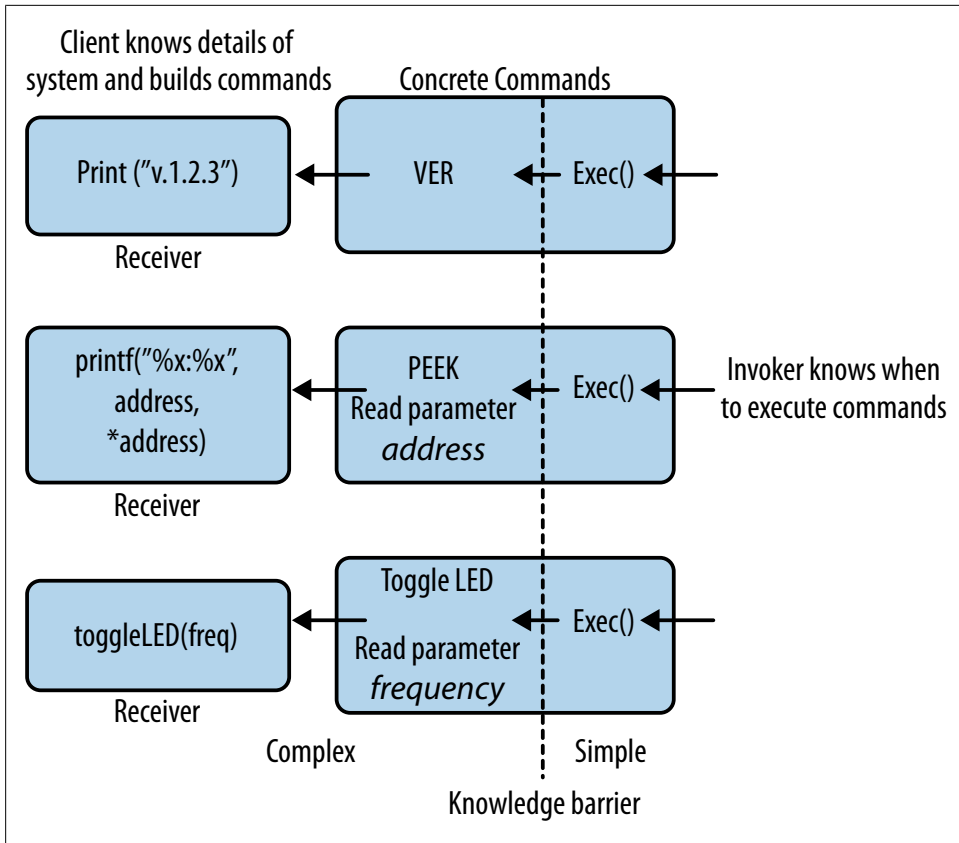


Figure 3-15. Command patterns knowledge barrier



Consider the client as protecting the secrets of the system by disguising all of the commands to look alike. The invoker will have to play by the rules or the system will know it is up to no good.

The command interface can be richer, possibly adding a log command, a help function, and an undo function. The invoker will know when to call these, but it will be up to the client to build each of these commands with the actual implementation.

Because the details are hidden, it is pretty straightforward to add a macro command that implements multiple commands. To do this, the client creates a list of the basic commands to be combined. When invoked, the macro calls the execute function on each of its subcommands (or undo or log). For example, a macro to print the version (ver) and read the data at an address (peek), would call the execute functions of those

commands. This provides a better layer of decoupling than if the macro called the receiver functions directly.

Handling Errors Gracefully

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 multi-layered 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 only be used 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 (i.e. an LED).

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) may 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. While 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've got 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 Handling Library

An error handling library is also a good idea. One way to implement one is to have each function return an error code. Instead of calling the function and checking the results like this:

```
error = FunctionFoo();
if (error != NO_ERROR) {
    ErrorSet (&globalErrorCode, error);
}
```

you'd call the function inside an error checking function:

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

The `ErrorSet` function would not overwrite a previous error condition if this function returned without a failure. 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.

Further Reading

If you want to learn more about handling hardware safely, reading schematics, and soldering, I suggest *Make: Electronics* by Charles Platt, published by O'Reilly (2009).

The step-by-step introduction is an easy read, though better if you follow along, components in hand.

To get a better look into hardware design, I suggest *Designing Embedded Hardware* by John Catsoulis, published by O'Reilly (2005).

If you are implementing code for a peripheral, get the latest datasheet version! Read the general description then skip to the text in the center, stopping at all timing diagrams. Don't forget to check the errata.

If you are evaluating a peripheral, find components that meet the bare minimum electrical and operating criteria. Check prices, lead times and future availability. Check performance characteristics (add requirements and desires to your criteria as needed). Make a prototype by mentally (or actually) implementing the code needed for this peripheral.

If you are getting used to a new processor, dev kits are the new user welcome mat for processors. Not only do they let you mock up the hardware, their getting started guides will give you the flavor of working with the whole environment: processor, compiler, debugger. This will help you a lot but it won't get you out of reading the user manual. Start with the introduction and wander where needed.

Learning to use your oscilloscope probably should be done with its manual, but I've found one useful, more general [write-up](#) that is very detailed. Alternatively, there are some videos on YouTube about setting up oscopes; sometimes watching someone do it is easier than reading about it.

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 if 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.

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 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 buzz words. But I do expect them to be able 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 made 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. 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 he has to do technically makes an engineer a much better engineer.

Outputs, Inputs, and Timers

As you've probably determined for yourself, projects don't always start off fully defined. Even the simple sounding subject of making an LED blink is not immune to product goals changing. In this chapter, we'll go through the phases of a project as the vision appears, changes, and finally crystallizes. Along the way, we'll focus on the input and output aspects of the system. From pin configuration registers to debouncing buttons to timers, this chapter will describe the most basic embedded concepts.

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, GPIO (general purpose I/O) and occasionally GIO (general I/O). The basic use case is straightforward:

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 low.

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. Atmel'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 NXP LPC 13xx User 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.

Starting with Registers

To do anything with an I/O line, we need to talk to the appropriate register. As described in [“Reading a Datasheet” on page 38](#), 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.

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 look them up. In C and C++, a bitwise-OR is `|` and bitwise-AND is `&`. The logical NOT operator (`!`) turns a 1 into a 0 (or a true into a false) and vice versa. The bitwise NOT (`~`) set each bit to its opposite.

```
register = register | (1 << 3); // turn on the 3rd bit in the register
register |= 1 << 3;           // same but more concisely
register &= ~(1 << 3);        // turn off the 3rd bit in the register
```

Enough review—if this isn't pretty obvious, look into bitwise operations and boolean math. You will need to know these pretty well to use registers.

An Introduction to Binary and Hexadecimal

Becoming familiar with basic binary and hexadecimal math will make your career in embedded systems far more enjoyable. Shifting individual bits around is great when you need only to modify 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)
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 (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. Since there are more digits in hexadecimal, we'll borrow some from the alphabet. In the meantime, 9 is just 8 + 1.

Binary	Hex	Decimal	Remember this number
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 twelve.
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 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, mapping 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 $(0x8 \ll 4)$. A 16 bit word is made up of two bytes, so 0x1234 is $(0x12 \ll 8) + (0x34)$. A 32 bit word is eight characters long in hex but ten characters long in decimal.

Since memory is generally viewed in hex, some values are used to identify anomalies in memory. In particular, expect to see (and use) 0xDEADBEEF as an indicator (it is a lot easier to remember in hex than in decimal: 3735928559). 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.

Set the Pin to be an Output

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 (i.e. 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 4-1](#)).

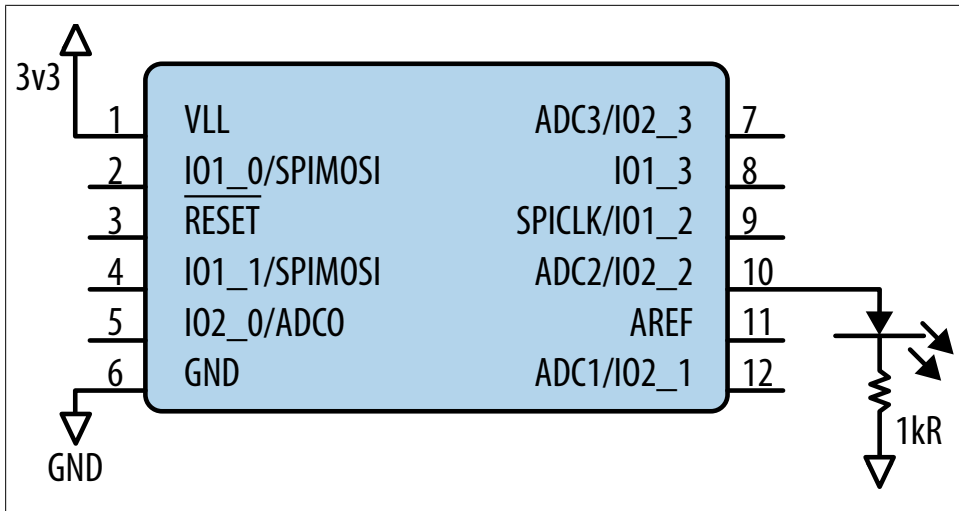


Figure 4-1. Schematic of processor with LED attached

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 “SPICLK/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 hard code the result:

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

However, please don't do that.

The processor or compiler vendor will almost always provide a header 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:

LPC13xx processor

```
LPC_GPIO1->DIR |= (1 << 2); // set IO1_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 relatively atomic chunks. If you read the value, modify it, 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 may also 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.

LPC13xx processor

```
LPC_GPIO1->DATA |= (1<<2); // IO1_2 high
```

MSP430 processor

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

ATtiny processor

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

The header file provided by the processor or compiler vendor shows how the raw addresses get masked by some programming niceties. In the LPC13xx.h, the I/O registers are accessed at an address through a structure (I made some reorganization and simplifications to the file):

```
typedef struct
{
    __IO uint32_t DATA;
    uint32_t RESERVED0[4095]; // The same data appears at 4096 locations
                             // in the GPIO address space and 12 bits
                             // of the address bus can be used for
                             // bit masking. (See manual 7-4.1)

    __IO uint32_t DIR; // direction set for output, clear for input
    __IO uint32_t IS;  // interrupt sense (1 = interrupt pending)
```

```

    __IO uint32_t IBE; // interrupt on both falling and rising edges
    __IO uint32_t IEV; // interrupt event register
    __IO uint32_t IE;  // interrupt enable
    __IO uint32_t RIS; // raw status register
    __IO uint32_t MIS; // masked interrupt status register
    __IO uint32_t IC;  // interrupt clear (set bit to clear interrupt)
} LPC_GPIO_TypeDef;

#define LPC_AHB_BASE      (0x50000000UL)
#define LPC_GPIO0_BASE    (LPC_AHB_BASE + 0x00000)
#define LPC_GPIO1         ((LPC_GPIO_TypeDef *) LPC_GPIO1_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.



You may want to update a vendor header file to use this structured approach. Approach this free code as a starting point, not the final draft. Modify it as needed to work in your system.

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 ([“Starting with Registers” on page 76](#)).

LPC13xx processor

```
LPC_GPIO1->DATA &= ~(1<<2); // IO1_2 low
```

MSP430 processor

```
LPC13xx 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 pseudo code 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

```

Once you've programmed it for your processor, you should compile, load, and test it. You might want to tweak the timing 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, 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, verify its functionality is as expected.

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 chapter of the user manual. This section gives you a high level idea but 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.

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.

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 waiting for a nonexistent external device. Turn off interrupts and asserts. Make sure the watchdog is off (see [“Watchdog” on page 143](#)). Put the LED code as early as possible in the code to reduce the likelihood that any other code is freezing the processor.

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 right 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!)

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 a LED on by writing a zero rather than a one.

If the output still doesn't work, consider whether there is a hardware issue. Even in hardware, it can be something simple (installing LEDs backwards 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: 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 a bit later. The system went from a prototype board to a PCB. Somehow in the process, the pin number changed (to IO1_3). They need to be able to run both systems.

For this project it is trivially simple to fix the code, 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 hard coding 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:

```
#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 IO to be output
LED_REGISTER |= LED_BIT;      // turn the LED on
LED_REGISTER &= ~LED_BIT;     // turn the LED off
```

That might 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:

```
// 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
#ifdef COMPILING_FOR_V1
#include "ioMapping_v1.h"
#elif COMPILING_FOR_V2
#include "ioMapping_v2.h"
#else
#error "No IO 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.



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 together even so simple an 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.

While 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. We've seen, though, 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, each function could be broken out so that there are `IOSet(port, pin)` and `IOClear(port, pin)` functions. 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`.

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, 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

As shown in [Figure 4-2](#), XOR is often represented as a Venn diagram where the overlapping section is not covered.

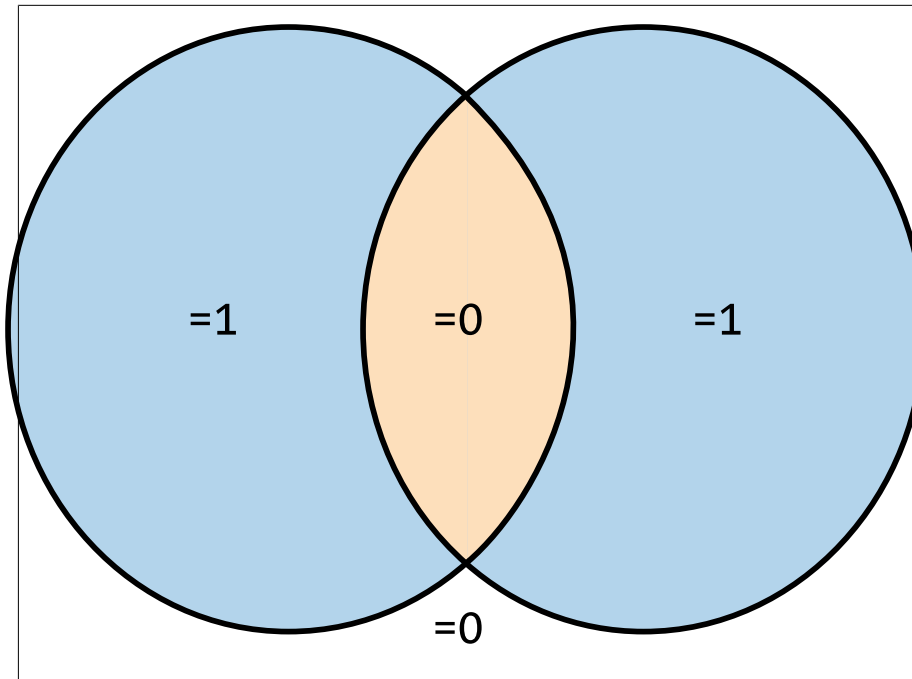


Figure 4-2. XOR Venn Diagram

XOR has some nifty applications in computer graphics and finding overflows (math errors). You can also toggle our LED on and off using XOR:

```
register = register ^ (1<<2);
```

Note that $(1 \ll 2)$ would always be a 1 in the table. If it is Input A, we are only using the bottom half of the chart. If the register (Input B) already has that pin set, what would the output be?

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 `IOWrite/IOWrite/IOWrite` 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:

```
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, its only dependency being the processor header. In the middle (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).

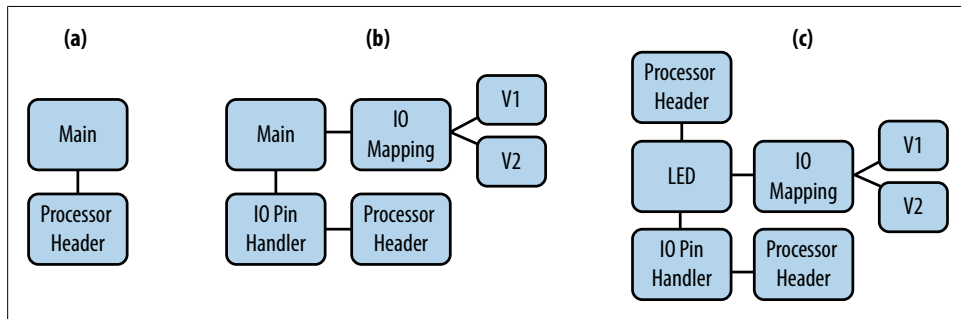


Figure 4-3. Comparison of architectures

Facade Pattern

As you can imagine, our I/O interface is going to get more complex as the product features expand. (Currently we've got 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 “[From Diagram to Architecture](#)” on page 16, we saw the adapter pattern, which is a more general version of the facade pattern. While 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 leaving the facade intact.

A facade for our blinking LED would hide the idea of I/O pins from 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

```
LEDBlink()
```

 Blinks the LED

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. For now, if 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 is I/O port 2, pin 2 denoted 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.

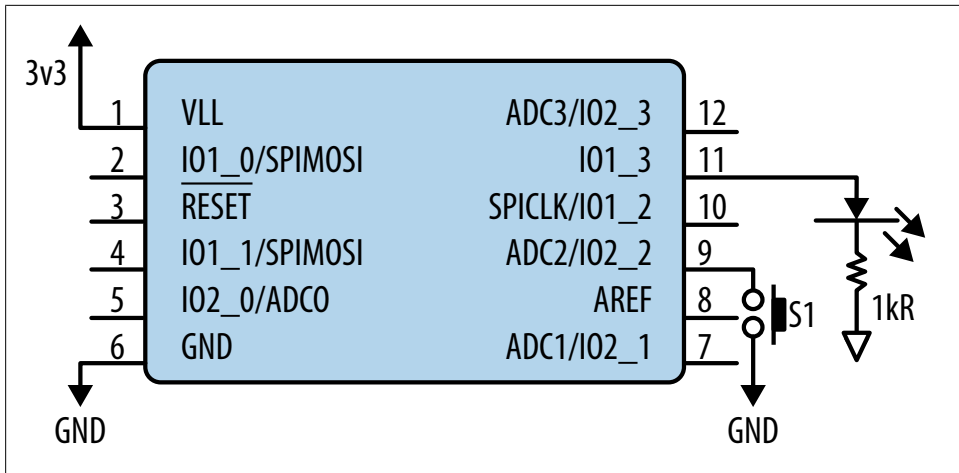


Figure 4-4. Schematic with LED and Button

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 depends on your processor (and possibly on the particular pin). Some 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.



Inputs with internal pull-ups take a bit of power, so if your system needs to conserve a few micro amps, you may end up disabling the unneeded pull-ups.

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 output. Verify it is not part of another peripheral.
3. Configure a pull-up explicitly (if necessary).

Once you've got 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 false (low):

```
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 if the button is currently pressed. The interface could be:

void ButtonInit()

 Calls the I/O initialization function for the button

boolean ButtonPressed()

 Returns true if the button is down

See the architecture in [Figure 4-4](#). 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.

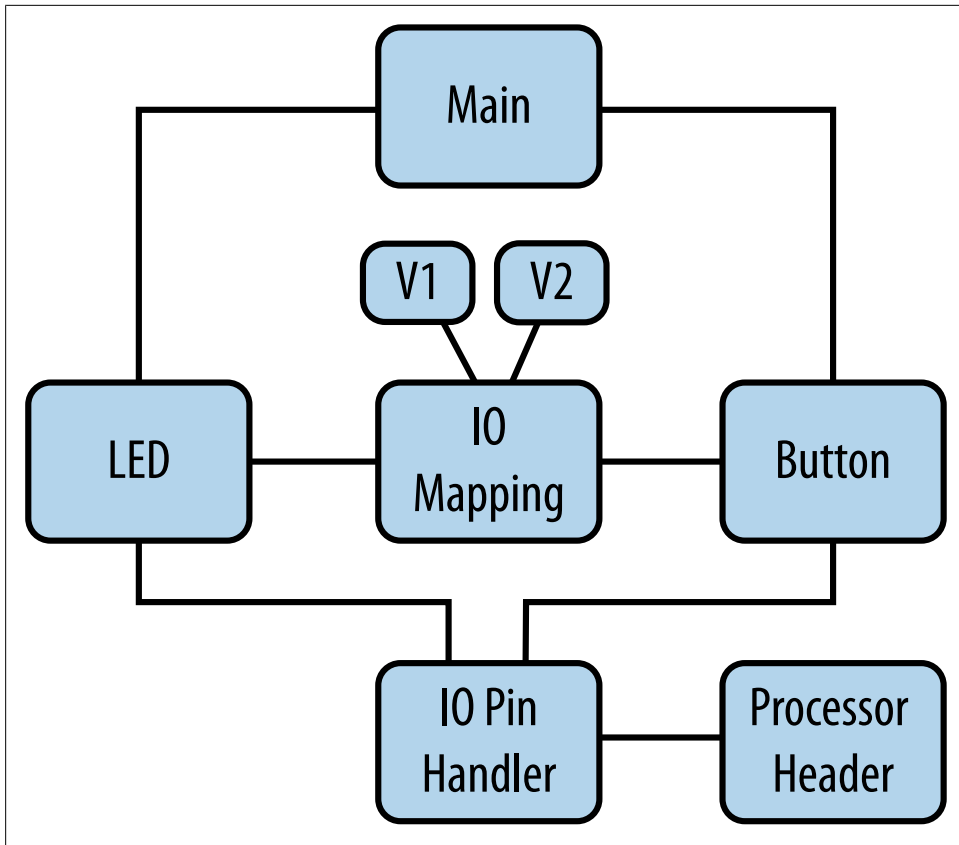


Figure 4-5. Architecture with Button

At a higher level, there are a few ways to implement the main function.

```
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.



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

To decrease the response time, we could constantly check to see if 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
```

These methods both check the button to determine if 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.

With interrupts, the main loop could be simpler.

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

The code to turn off the LED could be handled in the interrupt. 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 for an embedded system to 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 it has (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 simply 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.

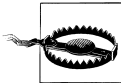
Interrupt on a Button Press

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 of 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
    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 part (a) 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.



To check a global variable accurately in this situation, you'll need the `volatile` C keyword, which you may never have needed before 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 volatile. If your code works fine without optimizations and then fails when optimizations are on, check that the appropriate globals and registers are marked volatile.

Configuring the Interrupt

Setting a pin to be an interrupt is usually separate from setting the function to set a pin to be an input. Although both are part of initialization, you should save the complexity of interrupt configuration for the pins that require it.

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

```
IOConfigureInterrupt(port, pin, edge or level, rising edge/high or falling edge/low)
```

Configures a pin to be an interrupt. Some systems also provide a parameter for a callback, which is a function to be called when the interrupt happens; other systems

will hardcore 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 may be per-bank), the processor may have a generic I/O interrupt, 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 interrupting on the glitches at the start and end of the button press.

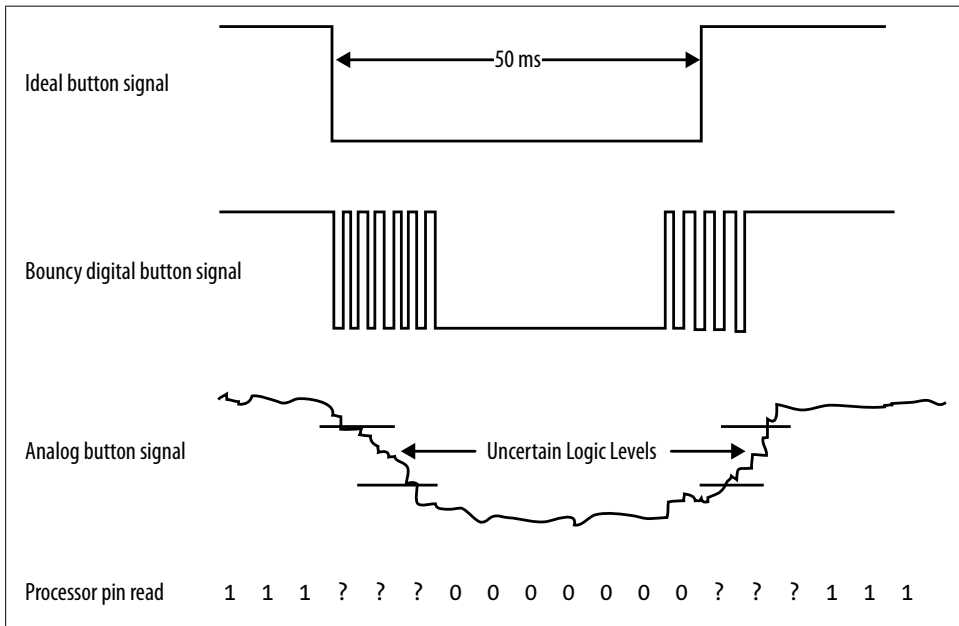


Figure 4-6. Different views of button signals

Switch bouncing is due to an inductive effect caused by the switch as the state of the signal transitions from one to another, and hence there is an associated change in the electric field. The switch is, in effect, acting as an inductor. Coupled with transmission

line effects in the signal trace, this is the source of the anomaly. It is not a mechanical "bounce," although this is commonly and incorrectly taught in many universities. The confusion arises because the inductive ringing is known as signal "bounce" and many people misinterpret this term. The switch doesn't physically bounce. You can drive the contact home with significant force (no mechanical bounce possible) and still get ringing. You will also get ringing/bouncing when a contact is electrically (rather than mechanically) toggled.

Figure 4-6 shows an analog view of what could happen when a button is pressed and only slowly goes into effect. Whether you have a bouncy or switch bouncing button, there are parts of the analog signal where the signal is neither high nor low, but somewhere in between. This leads to indeterminate values of the digital signal. 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.

Debouncing can eliminate the spurious edges. While it can be done in hardware or software, we'll focus on software. See Jack Ganssle's excellent web article on debouncing (in the references at the end of this chapter) for hardware solutions.



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 if you need debouncing may 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.

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 5 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 50ms. (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 12.5ms when pressed or released. If the goal is to respond to a button held down for 50ms or more, we can sample at 5ms (200Hz) 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. What you choose depends on the cost of being wrong (annoyance or catastrophe?) and the cost of doing it better (developer time and processor cycles).

In the previous 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
```

In this pseudo-code, 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 got the button press handled, 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 number button presses = 0, toggle blue LED
If number button presses = 1, toggle red LED
If number button presses = 2, toggle green LED
```

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); the latter means the LED function will need to map the color to the I/O pin each time it toggles the LED (which wastes 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 main 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 which LED

  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 if the `change which LED` 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 (like which LED is to be used). We'll need to abstract the LED code from the I/O pin by passing the I/O handler as a parameter to the LED code, eliminating any

direct dependence that the LED subsystem has on the I/O subsystem. This is called *dependency injection* because you inject the dependencies into the code, generally during configuration (when the button is pressed, the I/O to use is changed) instead of during normal operation.

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 an 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 hard coding the dependency at compile time. This technique allows you to compose the way the system works at run time.

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.



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

This is a very powerful technique, particularly if your LED module did something a lot more complicated, for instance, it outputs 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 speed of blinking 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: 13 times per second (Hz), 17Hz, and 20Hz.

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 see if marketing can accept that.

Timer Pieces

In principle, a timer is a simple counter measuring time by accumulating the 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 set the frequency of the timer, you will need to determine the clock input. This may be your processor clock (aka *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 short hand 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 LPC13xx, MSP430, and ATtiny processor families. Systems with these processors could have stats like:

- NXP LPC1313 (Cortex-M3), 32-bit, 72MHz
- Texas Instrument MSP430 G2201, 16-bit, 16MHz
- Atmel ATtiny45, 8-bit, 4MHz

That last number is the processor clock, and 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 the magic of PLL, your processor speed may much be faster than an onboard oscillator. PLL stands for *phase lock loop* which is the way a processor can multiply a slower clock (i.e. a slow oscillator) to get a faster clock (a processor clock). Since 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. While 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 4MHz. We want the LED to be able blink at 20Hz, 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 “[System statistics](#)” on page 98). However, the chip designers recognized this issue and gave us another tool: the prescale register, which divides the clock so that the counter increments at a slower rate.

The effect of the prescale register is seen in [Figure 4-7](#). The system clock toggles regularly. With a prescale 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 continue counting up or reset.

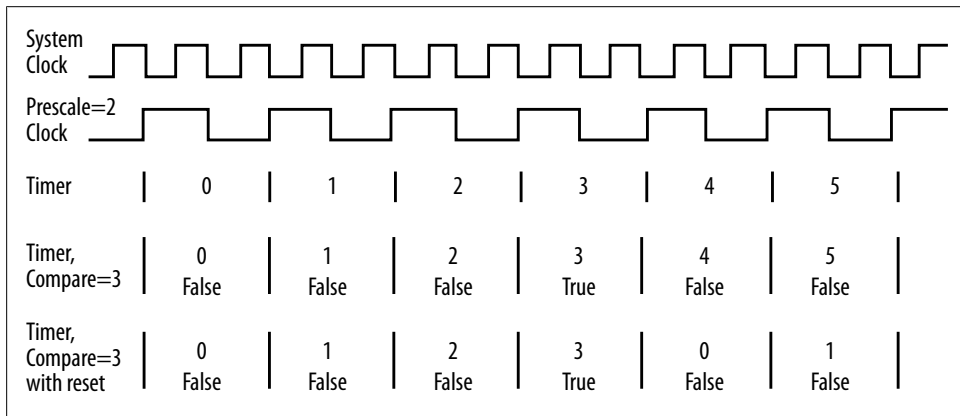


Figure 4-7. Timer prescaling and matching

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

Timer counter

This holds the changing value of the timer, the number of ticks since the timer was last reset.

Compare (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

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:

- Interrupt (or not)
- Stop or continue counting
- Reset the counter (or not)
- Set an output pin to high, low, toggle, or nothing

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.

Prescale register

As shown in [Figure 4-7](#), this divides the clock so that it runs more slowly, allowing timers to happen for relatively rare events.

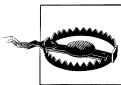
Control register

This sets the timer to start counting once it has been configured. The control register also often 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 will generally guide you through setting up each of these registers. Your processor user manual may give the registers slightly different names.



Instead of a compare register, your processor might only allow you to trigger the timer actions 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

Timers are made to deal with physical time scales, so you need to relate them from a series of registers to an actual time. Remember that the frequency (for instance, 14Hz) is inversely proportional to the period.

The basic equation for the relationship between the timer frequency, clock input, prescaler, and compare register is:

$$\text{timerFrequency} = \text{clockIn} / (\text{prescaler} * \text{compareReg})$$

This is an optimization problem. You know the `clockIn` and the goal `timerFrequency`. You need to adjust the prescaler and compare register until the timer frequency is close enough to the goal. If there were no other limitations, this would be an easy problem to solve.

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 (i.e. 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 only have 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 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, 4MHz system clock, and goal frequency of 20Hz, 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.

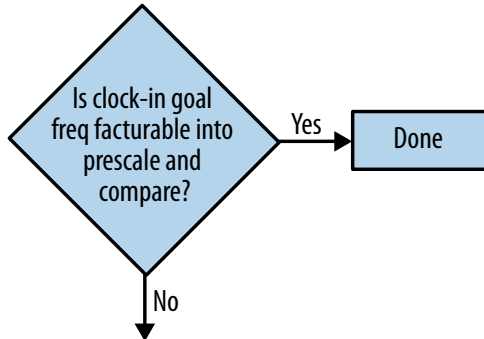
- The compare register has to lie between 0 and 255 (because the timer register is eight bits in size).
- The prescaler on the ATtiny45 is 10 bits, so the maximum prescaler is 1023. (The size of your prescaler may be different.)

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.

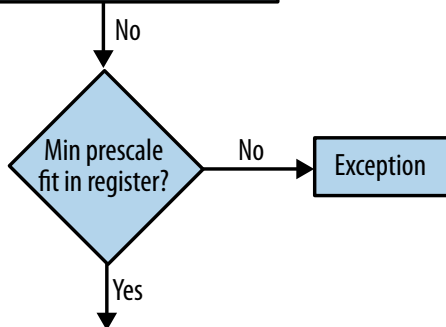
There are several heuristics for finding a prescaler and compare register that will provide the timer frequency needed (see [Figure 4-8](#)).



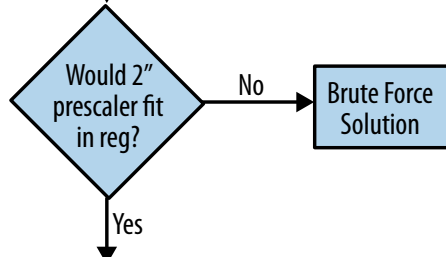
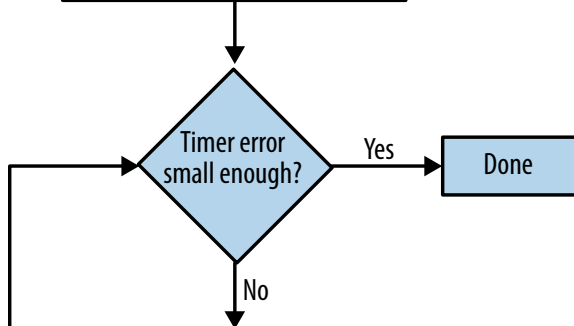
I asked two math professors how to solve this problem in a generic manner. The answers I got back were interesting. The most interesting part was learning that this problem is NP complete for two reasons: integers are involved and it is a nonlinear two variable problem. Thanks Professor Ross and Professor Patton!



Determine min prescale.
Round up to nearest integer.



Calculate compare value.
Round to nearest integer.



Prescaler * = 2
Find compare reg closer to integer given prescale + or -1.

We can determine the minimum prescaler by rearranging the equation and setting the compare register to its maximum value:

$$\begin{aligned}\text{prescaler} &= \text{clockIn}/(\text{compareReg} * \text{timerFrequency}) \text{ minPrescaler} \\ &= 4\text{MHz}/(255 * 20\text{Hz})\end{aligned}$$

Unfortunately, the resulting prescaler value is a floating point number (784.31). If you round down (784), the timer value will be above the goal. If you round up, you may be able to decrease the compare register to get the timer value to be about right.

In this case, we end up with a timer of 19.98Hz which is an error of less than a tenth of a percent off. 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:

$$\text{prescaler} * \text{compareReg} = 4\text{MHz}/20\text{Hz} = 200,000$$

This is a nice, round number, easily factored into 1000 (prescaler) and 200 (compare register). This is the best and easiest solution to optimizing `prescaler` and `compareReg`: determine the factors of $(\text{clockIn}/\text{timerFrequency})$ and arrange them into `prescaler` and `compareReg`. However, this requires the $(\text{clockIn}/\text{timerFreq})$ 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.

We can see this as we move along to another blink frequency requested by marketing (17Hz).

$$\text{prescaler} * \text{compareReg} = 4\text{MHz}/17\text{Hz} = 235294.1$$

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 above 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:

$$\text{error} = 100 * (\text{goal frequency} - \text{actual}) / \text{goal}$$

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 1023 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 timer 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 (i.e. Matlab or Excel) and brute force try out the options as shown in [Figure 4-7](#). 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 timer frequency. Round the compare register to the nearest whole number and calculate the actual timer frequency. This method led to a prescaler of 997 and 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.

$$\begin{aligned}
 \textcircled{1} \text{ Min Prescaler} &= \frac{\text{CLOCK INPUT}}{\text{GOAL FREQUENCY} \times \text{MAX COMPARE}} \\
 &= \frac{4\text{MHz}}{20\text{Hz} * 255} = 972.72 \\
 \textcircled{2} \text{ Max Prescaler} &= \frac{4\text{MHz}}{20\text{Hz} * 1} = 200,000 \quad \text{This is only 10-bit register (max 1023)} \\
 \textcircled{3} \text{ For each prescale value 973 to 1023} \\
 \text{Compare} &= \frac{\text{CLOCK INPUT}}{\text{GOAL FREQUENCY} * \text{PRESCALE}} = \frac{4\text{MHz}}{20\text{Hz} * 973} = 254.9 \\
 \text{Compare} &= \text{ROUND}(\text{COMPARE}) = 255 \\
 \text{Actual output frequency} &= \frac{\text{CLOCK INPUT}}{\text{PRESCALE} * \text{COMPARE}} = \frac{4\text{MHz}}{973 * 255} = 16.99 \\
 \text{Error \%} &= 100 * \frac{\text{ABS}(\text{GOAL-ACTUAL OUT FREQUENCY})}{\text{GOAL OUTPUT FREQUENCY}} \\
 &= 100 * \frac{(17 - 16.99)}{17} = 0.03\% \\
 \textcircled{4} \text{ Find PRESCALE and COMPARE REGISTER} \\
 &\text{pair with least error}
 \end{aligned}$$

Figure 4-9. Brute force timer solution

A Long Wait between Timer Ticks

Brute force works well for 17Hz, but when you get the goal output of 13Hz, 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 13Hz, we could have a 26Hz timer and toggle the LED every other time the interrupt was called. This method is less precise because there may be delays due to other interrupts.

Using the Timer

Once you have determined your settings, the hard part is over. There are a few more things to do:

- Remove the code in the main function to toggle the LED. Now the main loop will only need to have a set of prescale and compare registers to cycle through when the button is pressed.
- 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 (PWM)

Market research has shown that potential customers are bothered by the brightness of the LED. Marketing wants to try out different brightness settings (100%, 80%, 70% and 50%) using the button.

A timer is a set of pulses that are all alike. In PWM, the pulses' widths change depending on the situation. So a timer signal is 50% up and 50% down (this is known as 50% duty cycle). But 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 pulled low. The duty cycle represents the average value of the signal as shown by the dashed line in [Figure 4-10](#).

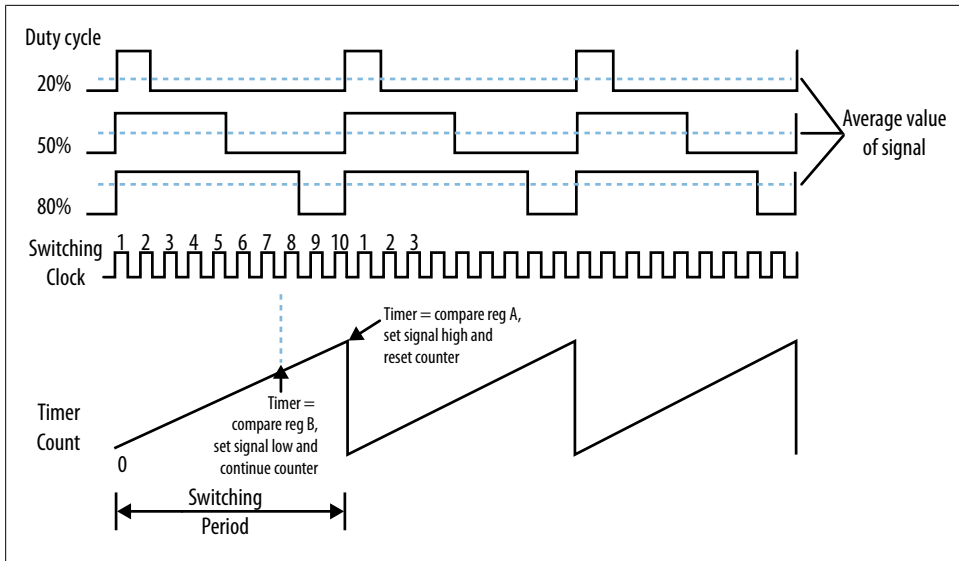


Figure 4-10. PWM duty cycles: 20%, 50% and 80%

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.

Given the timer from the previous chapter, we could implement a PWM with an interrupt. For our 20Hz LED example, we had a compare register of 200 so that every two hundred ticks, the timer would do something (toggle the LED). If we wanted the LED to be on 80% of the time with a 20Hz timer, we could ping-pong between two interrupts that would set the compare register at every pass.

1. Timer interrupt 1:
 - a. Turn on LED.
 - b. Set the compare to 160 (80% of 200).
 - c. Reset the timer.
2. Timer interrupt 2:
 - a. Turn LED off.
 - b. Set the compare register to 40 (20% of 200).
 - c. Reset the timer.

With a 20Hz timer, this would probably look like a very quick series of flashes instead of a dim LED. To alleviate that, increase the frequency at which the timer switches the LED state (aka the switching frequency). 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) which 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, main doesn't control the LED directly at all.

While 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 often need to modify the duty cycle. 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 (8Hz), 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 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, but 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 internally release a development version. It will help you find the removed features after you prune the code for shipment.

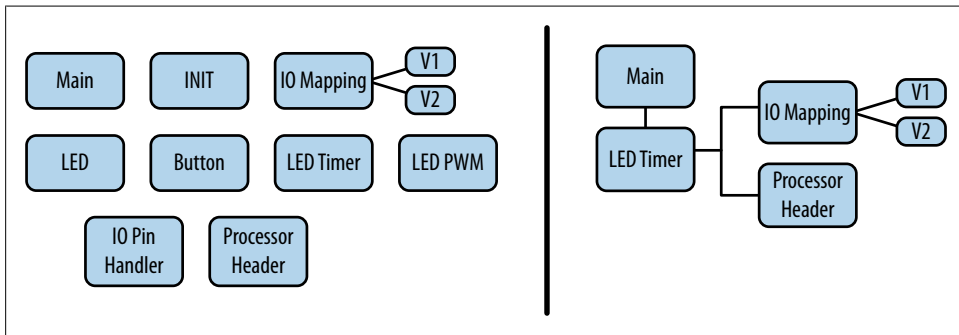


Figure 4-11. Comparing spaghetti prototyping code with a simpler design

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 tend to like readable files even if it means a few initial bugs but I can 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 separated 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

(These are kind of a mess, I'm not sure what really needs to be here other than the Ganssle pointer which could be moved up to be a footnote. Help!)

1. [A Guide to Debouncing](#): 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.
2. LPC13xx preliminary user manual (UM10375), Rev. 00.07, 31 July 2009.
3. MSP430 430x2xx Family User's Guide (slau144e.pdf), 2008.
4. Atmel user manual: 8-bit Microcontroller with 2/4/8K Bytes In-System Programmable Flash (ATtiny25/V, ATtiny45/V, ATtiny85/V), Rev. 2586M-AVR-07/10.
5. Atmel application note: AVR 130: Setup and Use the AVR Timers

Interview question: Waiting for a register to change

What is wrong with this piece of code?

```
void IOWaitForRegChange(unsigned int* register, 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 time out or error handling.

If an interviewee flails, pointing out non-critical 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).

Task Management

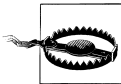
In [Chapter 2](#), we looked at different ways to break up a system into manageable chunks. That chapter described the what and why of design; this chapter covers the how. Once you've identified the pieces, getting them all to work together as a system can be daunting.

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 going to only 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” on page 145](#)).

Tasks

When you turn on your computer, if you are like I am, you load up the email program, web browser, and compiler. Several other programs start automatically (such as my instant message client). Each of these programs runs on the computer, seemingly in parallel even if you've only got one processor.



Three words that mean slightly different things, but that overlap extensively, are sometimes used interchangeably. A *task* is something the processor does. A *thread* is a task plus some overhead such as memory. A *process* is usually a complete unit of execution unit 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 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 things your system should do and chooses which one it does right now.

Without an operating system, you are going to need to do the scheduling yourself. The very 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 to 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?

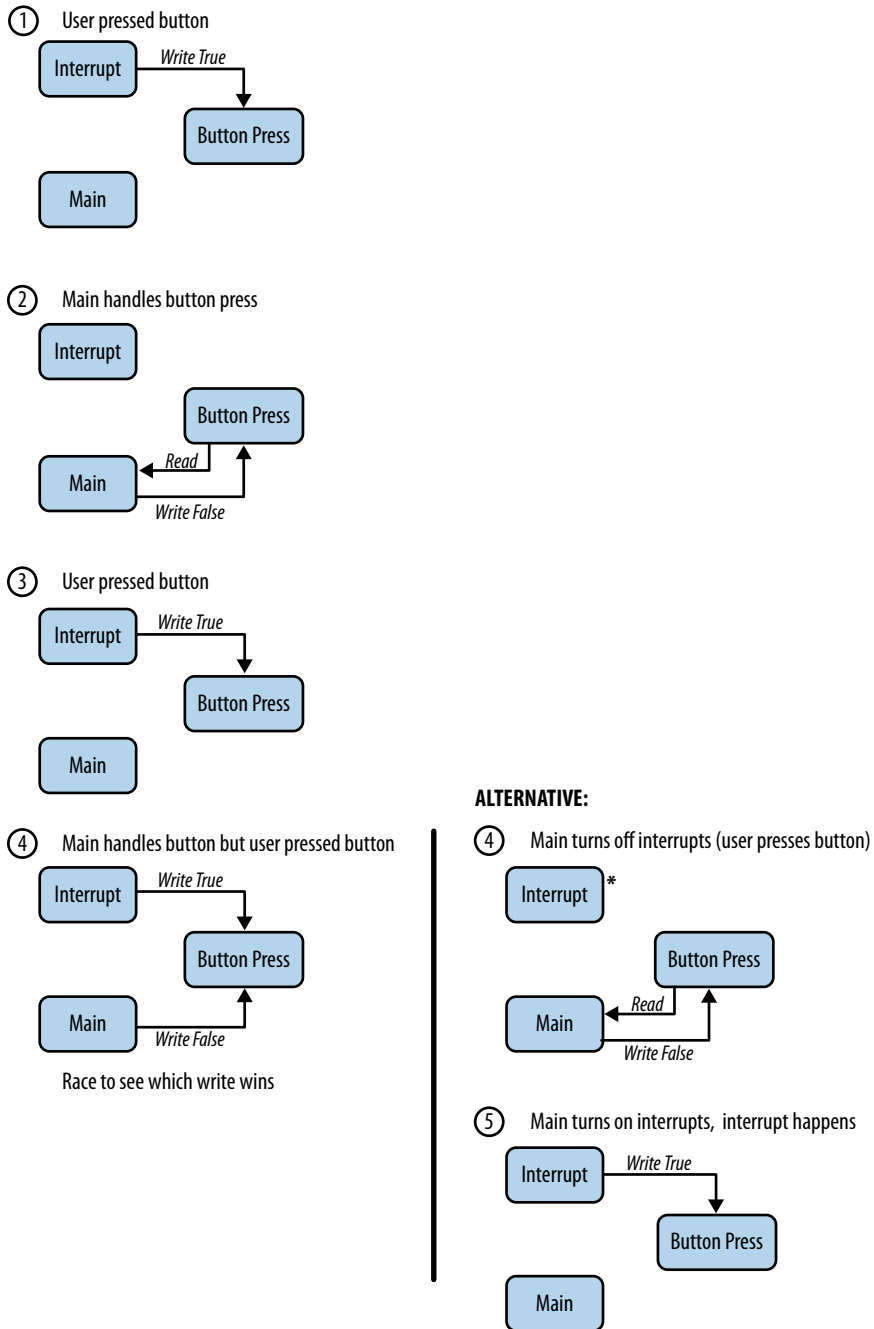


Figure 5-1. Race condition in shared memory

The answer depends on the precise timing of what happens. The uncertainty is the 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 the 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). While only a slight annoyance to a user in this 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*.

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.



Operating systems have heavier weight mutexes called semaphores that can handle situations where a thread or process can be preempted (forcibly changed by the scheduler).

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). While the flexibility can be very useful, it is possible to create a lot of trouble for yourself. Figure 5-2 shows the typical operating systems definition of priority inversion. It is okay 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.

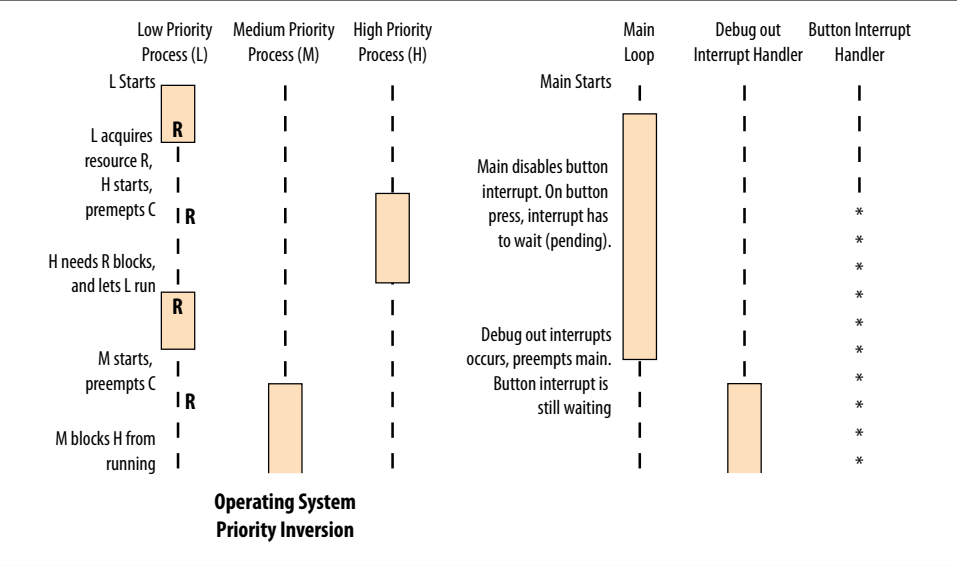


Figure 5-2. Priority Inversion

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. Now, 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 it interrupts the main loop, it is doing exactly that.

What is the most important thing for the processor to be doing? That is its highest priority. Or at least, it 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).

As we look at different ways of task management without an operating system, there are more ways to get into the position where the processor is inadvertently running a task that isn't the highest priority. Watch for this.

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.

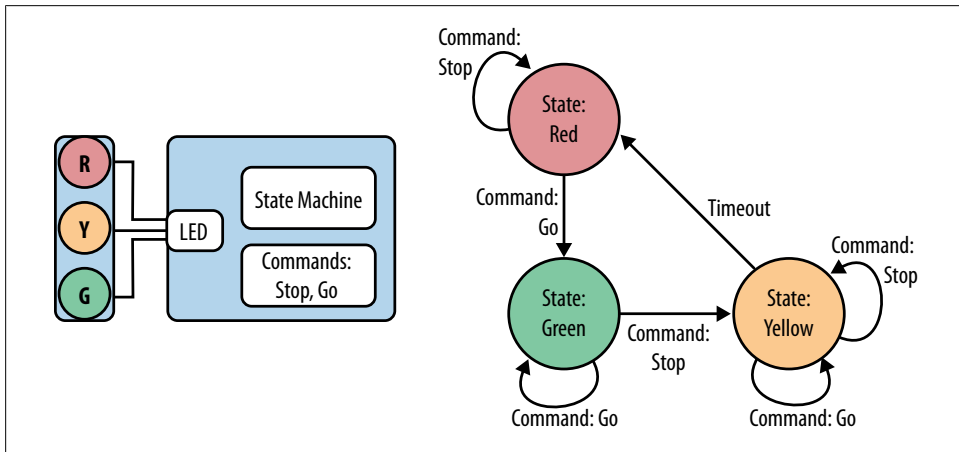


Figure 5-3. Stoplight System

We'll look at each element of this figure in the following section. The arrows in the diagram are just 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 a bit more complicated than a blinking LED and a button. [Figure 5-3](#) shows a stop light 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 transition 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 state (red, yellow, green), look at its connections. The stop and go commands should only happen 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 them here was to ignore them, putting a loop back to the associated state.

State-centric State Machine

Most people think of state machines 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 timeout)
            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 (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:

```

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 leave the code for each state simpler, and similar for almost all states. For instance, the green light state would look like:

```

case (green light):
    if (green light not on) turn on green light
    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 one if inter-state dependencies are there for a reason. State transitions are sometimes intertwined with state actions in ways that makes it more cumbersome to separate them than to keep them together.

Event-centric

Another way to implement a state machine is to turn it on its side by having the events control the flow so that 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 may still 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 (like checking for a timeout in the yellow state). You could make housekeeping an event if your system needs regular maintenance, or 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 state may just 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 table in [Figure 5-4](#), the possible events are columns of the table and each state has a row. Each box therefore shows what action needs to occur when the system is in that state and the event occurs. Often, the action is simply to move to a new state.

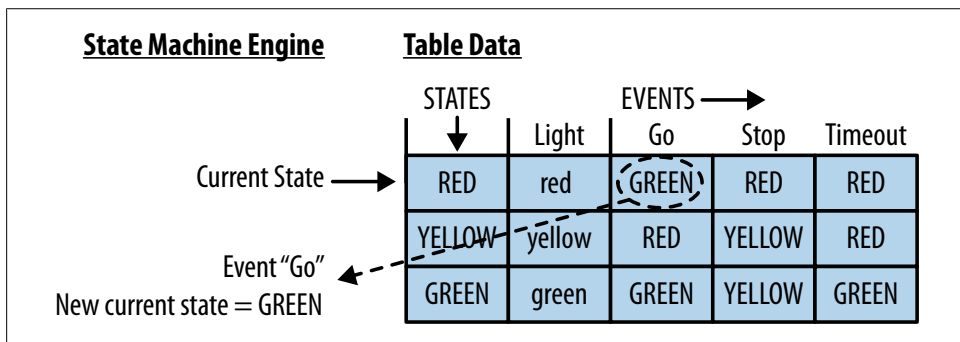


Figure 5-4. The state machine as a data table



When I worked on children's toys, they often offered 30 or more buttons (ABCs, volume, 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 helped with 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 two smaller, simpler pieces: a data table showing what state to go to when an event happens, and 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 spotlight 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)
{
    LightOff(currentState->light);
    currentState = currentState->go;
    LightOn(currentState->light);
}
```

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 like 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 let you see that the product handles events consistently.

The table can also 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 likely needs to return to one of the implementation mechanisms we showed earlier, oriented around control flow statements. For example, if we add some detail and error finding to the spotlight, 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 of the options I showed for a state machine offers the same functionality, even though the implementation is 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. 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 hows and whys of the events.

Interrupts can be kind of scary. They are one of the things that make embedded systems different from traditional software. Interrupts often seem to swoop in from nowhere to change the flow of the code. They can only call certain functions (and not usually the debug functions). Interrupts need to be fast, so fast that they are a piece of code that is still sometimes written in assembly. And bugs in interrupts are often quite difficult to find because, by definition, they occur outside the normal flow of code.

However, they are not the bogeyman they've been made out to be. If you understand a bit about what happens when an interrupt occurs, you'll find where they can be a useful part of your software design.

Looking back, I want you to remember that processors and interfaces are like software APIs (in [“Reading a Datasheet” on page 38](#)) and that function pointers aren't scary (in [“Function pointers aren't so scary” on page 66](#)). You'll need both of those ideas in your head as we walk through what happens when an interrupt occurs:

1. Interrupt request (IRQ) happens, inside the processor based on a peripheral, the software, or a fault in the system.
2. The processor saves where it was (the context).
3. The processor looks in the interrupt vector table to find the callback function associated with the interrupt.
4. The callback function (aka *interrupt service routine* (ISR) or interrupt handler) runs.

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.

But some 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 (brown out detection). Since these errors mean the processor can't run properly, they are often 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. Before we get to that, let's go through the process of interrupt handling in more detail.

An IRQ Happens

Usually an interrupt request happens because you've configured the interrupt. If it wasn't you, then your compiler'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).

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.

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 a NXP's LPC17xx processor which is middle of the road in its interrupt handling complexity (Atmel's ATmega is much simpler, the TI C2xxx is more complex).

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” on page 66](#). As noted in that section, accessing the memory address directly will make for illegible code. Most processor vendors and compiler vendors will give you a header files 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<<4); // disable timer 3 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 (0xE000E000) /*!< System Control Space Base Address */
#define NVIC_BASE (SCS_BASE + 0x0100) /*!< 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;
```



This header file structure can be built by reading the user manual, if you can't find some source code where someone else has done it for you.

This processor has separate clear and set registers as described in [“Writing an Indirect Register” on page 125](#), 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.

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.

A) Variable Style

	Contents of Variable								
1) Read	1	0	0	0	0	0	0	0	0x80
2) Modify	0	0	0	0	0	1	0	0	0x04
3) Write	1	0	0	0	0	1	0	0	0x80 0x04 = 0x84

B) Indirect Addressing

	Contents of Function Register								
1) Initial value	1	0	0	0	0	0	0	0	0x80
2) Set the third bit by writing 0x04 to set register	1	0	0	0	0	1	0	0	0x84
3) Clear the high bit by writing 0x80 to clear register	0	0	0	0	0	1	0	0	0x04

C) Reading register clears value

	Contents of Status Register								
1) Read status	0	0	0	0	0	0	1	0	0x02
*									
2) Read status	0	0	0	0	0	0	0	0	0x00

**Code did nothing between reads*

Figure 5-5. Methods for Setting Registers

3. Write the variable back into memory.

However, for a register that sets interrupt handling, this multi-step 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 set of *indirect* registers: a register that can set bits and a register that can clear bits, each of which acts on a third function register that is not directly writable (or memory mapped). To set a bit in the function register, you have to write the bit in the set register. To clear a bit in the function register, you write that bit in the clear register. In either register (set or clear), the unset bits don't do anything to the function register. (Side b of [Figure 5-5](#) gives an example of setting and clearing bits and the contents of the function register at each step.)

Sometimes, the function 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 function register can also be write-only without any way to be read. Some processors save memory space by having a register act as a function 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 function register using a *shadow variable*, a global variable in your code that you modify whenever you change the register. Instead of set and clear 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 5-5](#)). 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.

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 (LPC_TIM3). The user manual says that the match control register describes whether an interrupt should happen (yes) and whether the timer should reset (no) and start again (no).

```
LPC_TIM3->MCR |= 0x01;    // set the interrupt to occur
LPC_TIM3->MCR &= ~(0x02);  // timer should not reset after it expires
LPC_TIM3->MCR |= 0x04;    // timer should stop incrementing after it expires
```

Even though the first line of the 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 the interrupt on: a peripheral specific interrupt-enable like the one just shown, and a global interrupt enable like this:

```
NVIC->ISER[0] = (1<<4); // enable timer 3 interrupt
```

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 may want to wait to configure a few more things for our timer interrupt.

Multiple Sources for One Interrupt

Some processors have only one interrupt. The machinery necessary to stop the processor takes up space in the silicon. 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 (LPC_TIM3->IR & 0x1) { // this interrupt is due to a match at match reg 0
                          // on timer 3
```

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.

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 may be multiple causes. That leads us back to the question of priorities: which interrupt source will you handle first?

Interrupt Priority

As noted in “Scheduling and Operating System Basics” on page 111, 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 stayed 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 set the priority on a per-interrupt basis. See [Figure 5-6](#) for a snippet of the user manual for the LPC17xx.

6.5.12 Interrupt Priority Register 1 (IPR1 - 0xE000 E404)

The IPR1 register controls the priority of the second group of 4 peripheral interrupts. Each interrupt can have one of 32 priorities, where 0 is the highest priority.

Table 63. Interrupt Priority Register 1 (IPR1 - 0xE000 E404)

Bit	Name	Function
2:0	Unimplemented	These bits ignore writes, and read as 0.
7:3	IP_TIMER3	Timer 3 Interrupt Priority. 0 = highest priority. 31 (0x1F) = lowest priority.
10:8	Unimplemented	These bits ignore writes, and read as 0.

Figure 5-6. Priority register from LPC17xx manual

In the initialization code, we can set the priority level which is in the lowest eight bits of IPR1 according to the user manual:

```
NVIC->IPR[1] &= ~(0x000000FF); // set priority of timer 3 to be zero, highest priority
```

Of course, if you set all of the interrupts to the highest priority, the processor will choose which ones get set based on its own internal, documented criteria. (That also seems like a good metaphor for the life of a software engineer at a small startup.)

Nested Interrupts

Some processors allow interrupts within interrupts. Instead of priority being important only when an ISR is called, the priority is used to determine whether an interrupt can supersede 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.

When you clear the interrupt from the enable register, you disable that single interrupt from occurring. Some processors have another register that can turn off all interrupts (global interrupt enable/disable).

Non-Maskable 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 exceptions 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.

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 \(and heaps\)](#)” on page 225). 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 registers are saved so that the interrupt code can use them in its execution.

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.

These processor cycles are lost. If you had a processor that could do a hundred instructions a second (100Hz) 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. While 10 cycles is a decent interrupt latency (not great, not bad), 100Hz systems are rare. However, doing this math with a 30MHz processor, handling audio in an interrupt at 44100Hz with a 10-cycle latency uses 1.47% of the processor's cycles simply calling the interrupt handler.

Processor designers work to keep the interrupt latency low. One way to do that is to store the minimum amount of context. That 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 latency.

Some compilers will place limitations on the interrupt handling code you can write to minimize the latency, possibly limiting the number of variables you can have or the number of nested functions.

Calculating System Latency

Calling functions while in an interrupt is often 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 decreases. Going back to the 30MHz processor with its 44100Hz interrupt, if each interrupt uses 10 cycles for interrupt overhead and 10 cycles calling five short functions (each) which collectively use 275 cycles actually processing information, no other interrupt can be handled for at least 335 cycles ($10 + 5 \cdot 10 + 275$). The system latency is 11 microseconds ($335/30\text{MHz}$). Also note that the system spends nearly 50% of its time in the interrupt. Reducing the overhead of the interrupt will free up processor cycles for other tasks.

Reentrant Functions

Sometimes it is worth the overhead to call a function, though you've got to be careful about which ones are called. We've already seen some of the damage that a race condition can cause when interrupts and normal code try to share a global variable. However, functions which are called by interrupts must have an additional layer of protection.

A reentrant function is one that can be safely called multiple times, while it is running. This garden-variety swap function, for instance, is non-reentrant, because it has a temporary variable `t` whose value must not change before it is used to set the final `y` variable:

```
int t;
```

```

void swap(int* x, int* y) {
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
}

void isr(){
    int x = 1, y = 2;
    swap(&x, &y);
}

```

Functions that use static or global variables are non-reentrant. Our state machine with its internal state is non-reentrant. Many C++ objects are non-reentrant by virtue of their private data. Worse, some standard library calls are non-reentrant, including `malloc` (or `new`) and `printf` (any I/O or file function).

When the system saves the context, it does the bare minimum. The compiler can do more, but even that does not give you an entirely clean slate to work with. That is actually a good thing, because you usually want a global variable or two to signal the runtime code that the interrupt occurred. However, while in the interrupt, your code should not call any function that uses a global variable.



Count on interrupts to occur at the worst time. That way, you can design your system to minimize the havoc they can cause.

Get the ISR from the Vector Table

The third step to 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. 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. Really, the vector table is just a list of function pointers.

Initializing the Vector Table

The startup code (which probably came with your compiler) 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. That's still a bug, but at least then 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 preprocessor magic will make the compiler 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 LPC17xx processor, using the CodeRed compiler, we'd only need to name the ISR `TIMER3_IRQHandler` and the compiler would put the correct function address in the vector table. In some processors, such as Atmel's AT91SAM7S, you have to create a function and put it in the table more manually:

```
interrupt void Timer1ISR()
{
    ...
}

Void YellowTimerInit()
{
    ...
    AT91C_BASE_AIC->AIC_SVR[AT91C_ID_TC1] = // Set the 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 startup code, you don't usually need to know how to do it. However, in case you are curious, let's dig into the startup code for the LPC17xx for a moment:

```
_attribute__((section(".isr_vector")))
void (* const g_pfnVectors[])(void) = {
    // Core Level - CM3
    &_vStackTop, // The initial stack pointer
    ResetISR,    // The reset handler
    NMI_Handler, // The NMI handler

    ...

    TIMER2_IRQHandler, // 19, 0x4c - TIMER2
    TIMER3_IRQHandler, // 20, 0x50 - TIMER3
    UART0_IRQHandler,  // 21, 0x54 - UART0

    ...
}
```

First in the listing is a non-standard 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 files scripts ([“Linker Scripts” on page 205](#)), 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, 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.

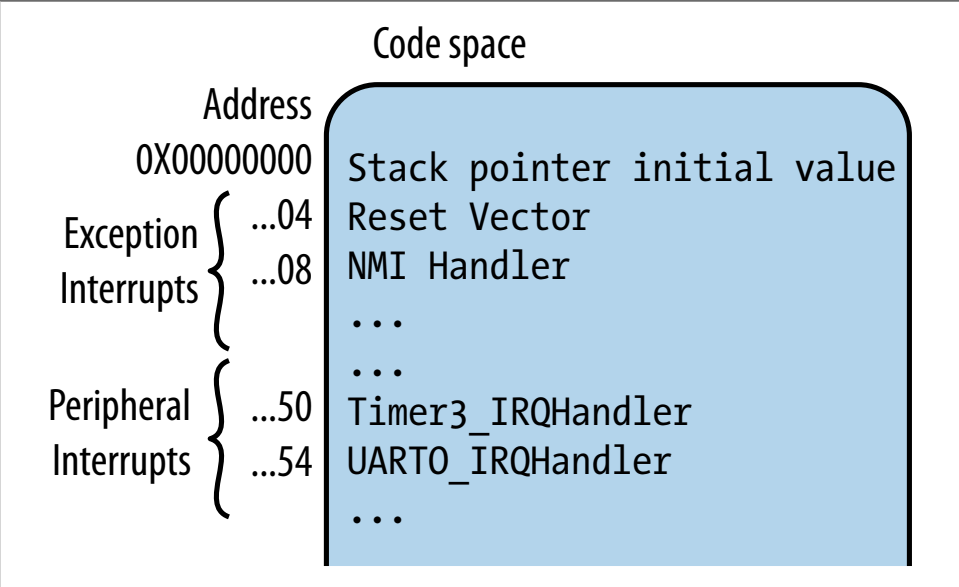


Figure 5-7. Vector table in memory

Each interrupt has an interrupt number. For our timer 3 interrupt, it is 20. When the interrupt happens, it is signaled by just this number: 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 address to the handler is found by multiplying the number by 4 ($20 \times 4 = 80$ or $0x50$) and looking at that slot in the vector table.

In this table, the compiler vendor has filled in all slots with dummy interrupt handlers that run infinite loops, to help the programmer catch misconfigured or spurious interrupts.

Calling the ISR

So far we've been going 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
                                         // cleared by normal code when event handled

void TIMER3_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 stop light controller is also taking pictures of red-light runners with a traffic camera, dispensing gimballs to good citizens who use the crosswalk, and trying to gain sentence. 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.

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. This is usually accomplished by reading a status register. As noted in [“Writing an Indirect Register” on page 125](#), this may have the side effect of clearing the bit.

Disabling Interrupts

Nested interrupts are allowed on the LPC17xx, 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 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've got 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've got 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.



Since some processors won't let you nest critical pieces of code, you'll need to be sure to avoid doing it accidentally. I recommend naming the functions in a way that indicates they turn off interrupts to avoid this issue.

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 level = disable interrupts
    do critical things
    enable interrupts(interrupt level)

CriticalFunction:
    interrupt level = disable interrupts
```

```
call HandyHelperFunction
do critical things
enable interrupts(interrupt level)
```

Since the helper function doesn't actually disable interrupts, it doesn't enable them either. The critical code remains safe throughout both functions.

Restore the Context

After your ISR has run to completion, 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.



It is 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 was a function, the things that the processor did to store the context would not get undone. So interrupts have a special 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 before the function call before continuing on its way.

This difference between a call and a jump is an important one, especially if you end up doing any assembly programming. Some alliteration may help you remember the difference: a call has context, a jump just goes. If you use `gotos` in your code, you are executing a jump.

Yes, people do use `gotos` in modern code. Used sparingly, they can act as exception handlers, cleaning up memory or a peripheral when things have gone catastrophically wrong during its use. They often replace code that starts out with a variable that tracks the error condition, getting checked at each stage in a series. When too many stages are preceded by `if !error do...`, the code morphs to just "goto an error handler" as soon as an error occurs, leaving the normal flow code unburdened by `if` statements.

If your 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.

When To Use Interrupts

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 time-out 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 is the requirement to handle a change on the line, the more an interrupt is appropriate for a solution.

Under that criterion, the button press we saw in [Chapter 4](#) that only needed a simple response within 50ms 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've got a lot of them. Interrupts also make your system less deterministic. One of the great things about not having an operating system* 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.

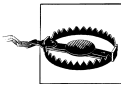
* Some real-time operating systems (RTOSs) are deterministic, usually the more expensive ones.

How Not to Use 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 (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.



When you get to [Chapter 10](#), 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. 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 just one subtlety worth mentioning: if you are polling and waiting for the hardware to do 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. While the amount of time in that tick varies, one millisecond tends to be a popular choice.

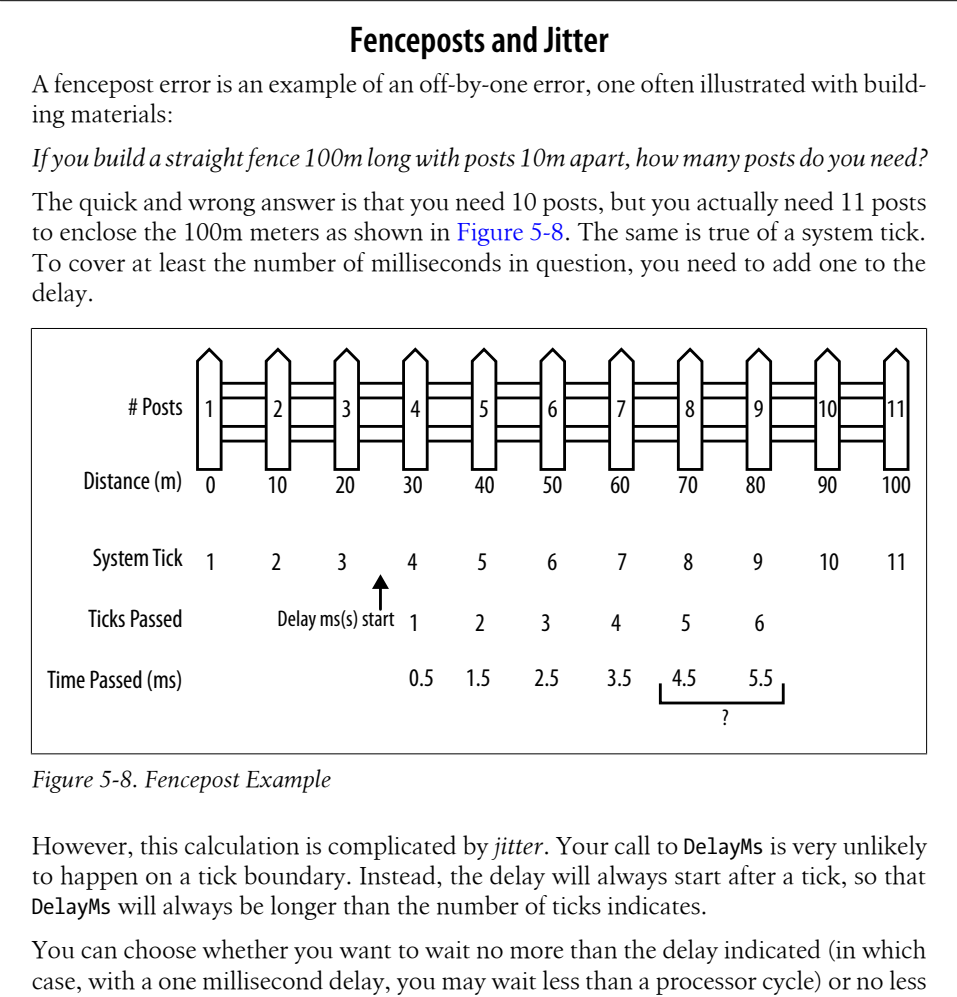


Ticks don't have to be one millisecond. If you've got a time that is important to your system for other reasons (e.g. you have an audio recording system that is running at 44100Hz), you may want to use that instead.

The tick is implemented with a timer interrupt that counts time passing. Yes, if we implemented the yellow light time out 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 “[Fenceposts and Jitter](#)” on page 139). Note that because of fence posting and jitter, `DelayMs` isn't a good measure of a single millisecond. However, if you want to delay ten or a hundred 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.



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 was finished. However, then we couldn't respond to any other commands. In this example that 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 1ms tick will make the clock roll over every 65.5 seconds. If need 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 1ms 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 get 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 + (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, 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).



At this point, you are starting to recreate 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. While 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.

```
run the scheduler after initialization is complete
LastScheduleTime = TimeNow()

while (1) {

    if TimePassed(LastScheduleTime) > ONE_SECOND
        run the scheduler, it will call the functions (tasks) when they are due
        LastScheduleTime = TimeNow()
}
```

We've already got the time based functionality worked out 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 just 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 timers. Upon finding an enabled timer, 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 timer. 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, other parts of your code subscribe to it. Sometimes the subscribers only request 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. Many embedded systems need to be self-reliant. They can't wait for someone to reboot the system when the software hangs. They may 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 though your system is stuck in an infinite loop. This defeats the purpose of the watchdog.

Setting the delay function (DeLayMs) to service the watchdog on the idea that the processor isn't doing anything else then.

You'll have scatter delay functions scattered around the code so the watchdog gets 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 hung. 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, 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 take aways here: 1) terminology for standard things changes, 2) language matters, 3) never let your CEO see your code.

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, since you are doing the work yourself, you need to know more about how OSs function so you can recreate the parts you need. There are many good OS books, but my favorite is the classic text book from Andrew Tannenbaum: *Operating Systems: Design and Implementation* (Pearson).

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* from Rodnay Zaks, 3rd ed. (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”).

Finally, the Arduino is a nifty and popular embedded platform, particularly for home projects. This [blog post](#) on setting up Arduino timer interrupts gives a great walk through of interrupts.

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 pseudo code. (During this time, I've drawn an intersection as in [Figure 5-9](#). I tend to draw this intersection on their piece of paper if I can).

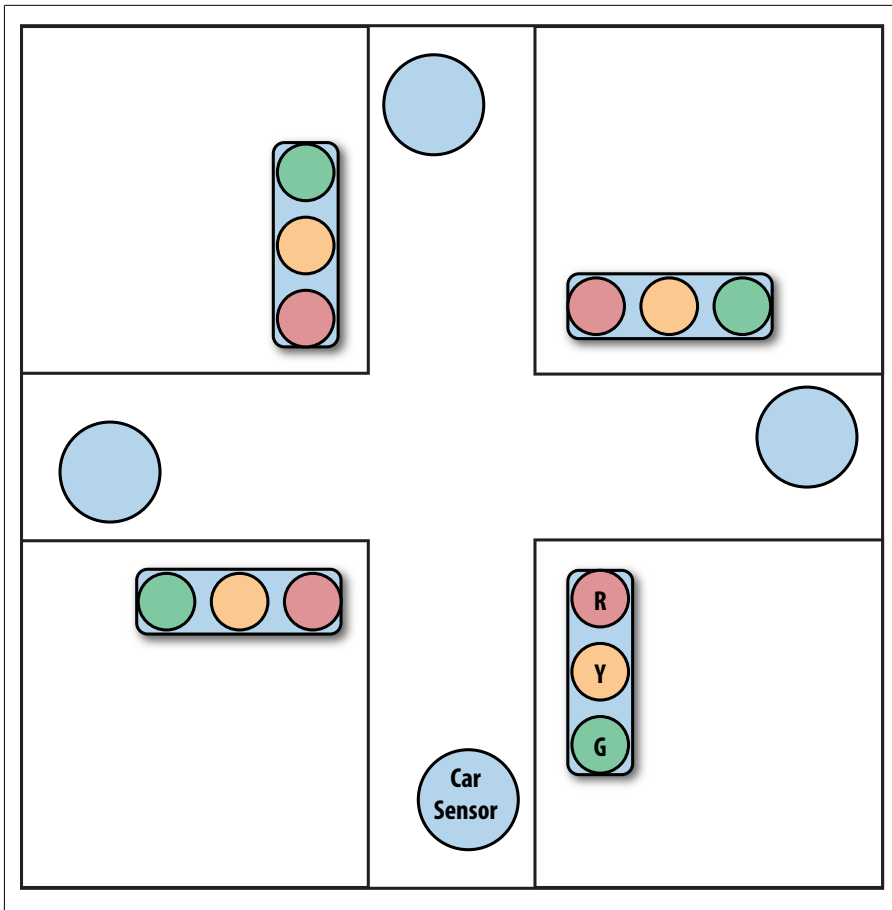


Figure 5-9. Intersection in a small city

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), some moniker that makes sense for the situation (First and Main)

or place on the page (right/left and up/down). Until she's put names on the diagram, 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 flow charts. 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 curve balls (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.)

Since 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?”).

Communicating with Peripherals

We've investigated heavily what goes on inside your processor during the course of this book. Now, before you can build a system, we need to consider what goes on in other components. Ultimately, the information you need are in datasheets, so if you're comfortable extracting information from them, you might be able to skip the first two thirds of this chapter, which cover fitting peripherals into embedded systems, comparing and contrasting some peripherals, and discussing common communication methods. The last third is more software-oriented, offering strategies for making them work well together. Don't skip that part.

The Wide Reach of Peripherals

A peripheral is anything outside your processor that it communicates with. Peripherals come in all shapes and flavors. Because your processor has direct access to memory, it barely counts as a peripheral (though it can be outside your chip). Sensors tell your system about its environment. Actuators act up on the environment. And displays interact with users.

I'm going to mention the most obvious of each of these so you get some ideas for your options. However, don't let my lack of creativity limit your ideas. And if you can't find what you want, wait six or eighteen months and someone will come along with it (yay for I2C tricolor LEDs!).

External Memory

We've already looked a bit at memory and how it is classified into *volatile memory* (lost when the system turns off) and *nonvolatile memory* (retained through power cycles). RAM is the most predominant form of volatile memory. It comes in different flavors; the software doesn't usually need to know the specifics.

On the other hand, most nonvolatile memory requires a software driver that recognizes the features of the memory style. As noted in [Chapter 3](#), the two more common subtypes

of this type of memory are EEPROM and flash. EEPROMs are generally smaller and their bytes can be written individually. Flash is made up of sectors that must be erased all at one time (the larger the flash, the larger the sector). Once a sector is erased, flash can be written one byte at a time.



The `volatile` keyword in C/C++ refers to something different from the idea of volatile memory. The keyword tells the compiler, “this variable may change outside the code that is currently running,” whereas volatile memory changes only when power is lost.

Once you've classified your memory as volatile or non-volatile, you'll need to a few more pieces of information about your external memory:

- How large is it? (Don't be fooled by memory sizes given in Mbits instead of Mbytes!)
- How long does it take to access the memory? This depends both on the communication bandwidth and the memory characteristics.
- How much do you need to erase at a time?
- How many times can it be rewritten?

Your product team may have a few other questions to help determine what kind of external memory you need:

- What is the price per bit?
- How large is the memory physically? (Or the density of the memory, the physical size per bit.)
- How much power does it consume? (RAM tends to be the worst, then flash, then EEPROM)

Once you choose what is right for you, memory is probably the easiest peripheral you will have to work with.

Buttons and Key Matrices

You've already seen buttons with a simple I/O interfaces (covered in [Chapter 4](#)). However, if you've got a bunch of buttons (e.g. a keyboard), you don't need one I/O line per button. You can matrix the inputs to get a lot more out of your I/O than you expect. There are two ways to implement a *key matrix*, depending on whether you need it to be cheap (row/column scan) or minimal I/O (*Charlieplexing*).

With a *row/column scan* you can implement an MxN matrix with M+N lines. So if you want to implement a 12 digit number pad, you could do a 3x4 matrix and use 7 lines (far fewer than the 12 lines it would take to do direct I/O). Of course, matrix input does require more complicated software to make it work.

In the electronics, each button is connected to one row and one column (and not an input pin to ground, as would be the normal I/O interface). On initialization, make all of the rows outputs and set them to be low. Then, make all of the columns inputs. As you are reading for button presses, set a row high, read the column's inputs, set the row low, and move on to the next row. A button that is pressed will be high when you read its column. You will need to map the row/column value to the button's actual meaning. [Figure 6-1](#) shows a snippet of a schematic with the row/column scan. Each circle is a button.

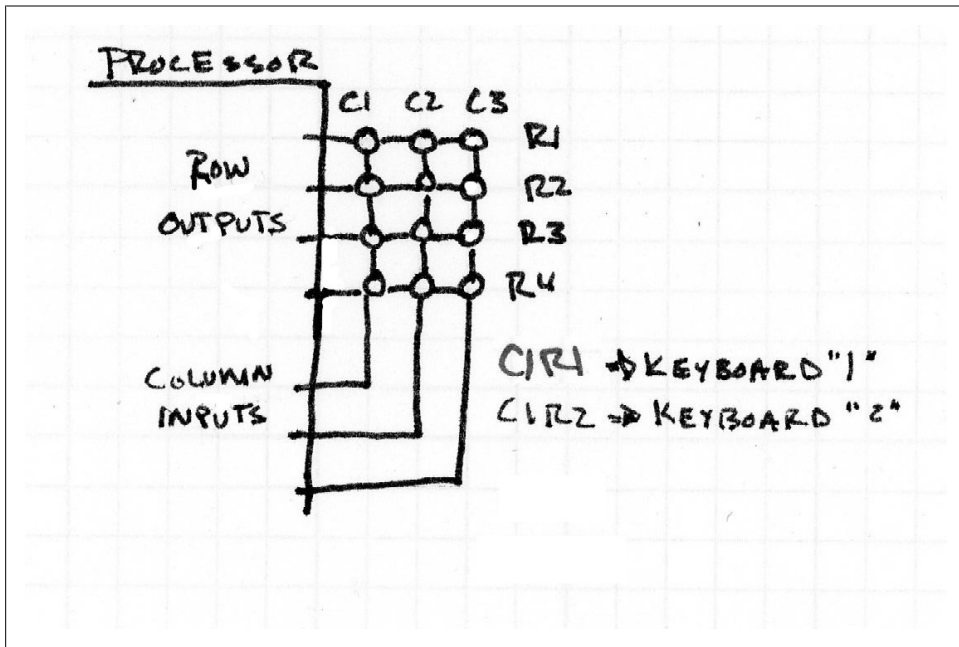


Figure 6-1. Row/column scan style key matrix

Note that matrixing the buttons means that they need to be polled. Although you can set up a timer interrupt to do that, you can't use a processor interrupt to tell you when an input has changed. The software needs to keep cycling through the rows.

This is also true of Charlieplexing, which is normally used with LEDs to make complex displays from only a few I/O pins. The same process can be used on inputs, but it requires more electronics (diodes) than the row/column matrix method. Instead of using N pins to receive N inputs in direct drive, or $M+N$ pins to receive $M \times N$ inputs in row/column, Charlieplexing lets you use N pins to get $(N^2 - N)$ inputs. So for the 12 buttons in a number pad, you can use a mere 4 I/O lines (and 12 diodes, fewer if you don't care about detecting multiple buttons pressed). While the software is more complicated than the row/column matrix, it is just a matter of taking it step by step. A [web](#)

[site about Charlieplexing](#) uses excellent animations to walk through how it works. (They also walk through the [row/column method](#) with good animations.)



Both of these methods may cause problems when multiple buttons are pressed simultaneously.

Row/column matrix and Charlieplexing can be used on outputs (LEDs) as well as inputs (buttons). The methodology is the same in each case, so once you wrap your head around matrices you'll find plenty of places to implement them.

Sensors

A button could be called a type of sensor. The methods used to talk to buttons are similar to those used with sensors, and the choices are similar (e.g. interrupts versus polling). However, buttons are either pressed down or not. Sensors can provide all sorts of data—high, low, 1.45, 10342.81, 12, 43, or out of range. What you read depends on your sensor.



Your system may have its own input sensors while serving as a sensor to another system.

Analog Sensors

If you have a microphone, like your ears, the audio it picks up is analog. You can use an ADC to move from real-world analog to software-friendly digital (ADC means analog-to-digital converter, sometimes called an A2D). Your software can do any number of things with the digital data: watch for events in the analog stream, filter it, or change it back into analog with a DAC (digital to analog converter, aka D2A). In analog form, the signal can be played through a speaker. [Figure 6-2](#) shows a simple system.

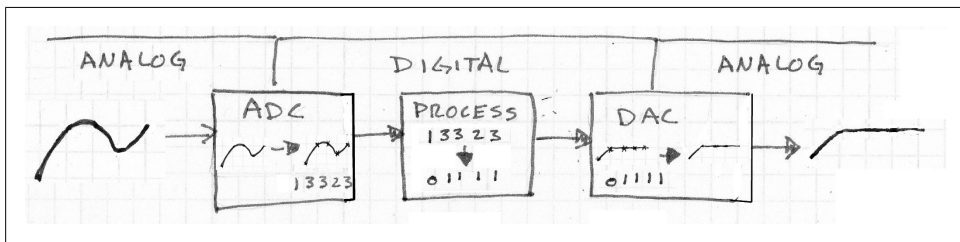


Figure 6-2. Analog to digital to analog system

The *signal* is the underlying thing you are looking at, whether it is currently digital or analog. *Noise* is what makes it hard for your software to see the signal properly. A goal

of the system is to make sure there is plenty signal compared to the noise giving you a high SNR (signal to noise ratio).

Signal processing is a software technique for changing data from analog to digital, and often doing something to make sense of the data. Sometimes the data gets transformed to look like something different. Often, this change converts the data to a combination of sine waves because these are easy to compute with. The sine waves are frequencies (think of radio station frequencies or musical pitches). So this conversion is called *conversion to the frequency domain*.

In the [Figure 6-2](#), the signal is reduced by dividing each integer sample by two. The resulting analog sound should be quieter (attenuated) but integer division isn't a good idea with such small numbers, so the signal morphs more than it should. (Receiving a higher signal input at the ADC or sampling more bits in the ADC would reduce this problem with integer math.)

This is a relatively silly example, because you could get a better attenuation effect without digitizing the input. More often, the data goes through multiple stages of processing. Some of these might include filtering (to reduce noise or enhance a particular signal characteristic), companding (attenuating loud sections and/or boosting quiet ones), and convolution (finding one signal hidden inside another).

Some processors are built with signal processing in mind (DSPs, aka digital signal processors). They have special processor instructions that work with the signal much faster than a general purpose processor does.

Of all of the embedded systems areas, signal processing is my favorite. However, it is a complete book unto itself. Well, two books, because you need to understand the math (Fourier is fun) and how to apply that to real-world problems (Fast Fourier Transforms!).

I haven't talked about sampling and the Nyquist frequency. That is because I haven't given you nearly enough information to be able to do anything useful with analog sensors. Either you already know this or you need far more than this brief orientation. I don't want you to think you are ready to go off and make a great analog system with this tiny section. I recommend my favorite signal processing books in [“Further Reading” on page 195](#).

I started this section with a microphone, but analog sensors come in all sorts of types—motion sensors (accelerometers, gyroscopes, magnetometers), weather sensors (wind, humidity, temperature), vision sensors (cameras, infrared detectors), the list goes on and on. You may only have five senses, but your system can have many more.

MEMS (microelectromechanical systems) sensors are a special type of sensor that uses a tiny (nanoscale) sensing element. While MEMS have given us more robust and a wider variety of sensors to play with, from the software point of view, MEMS sensors are the same as any other sensor.



I once heard an electrical engineer say, “All sensors are temperature sensors but some are better than others.” It is true. Sensors respond to ambient temperature even when they are supposed to be measuring something else. Expect to have to calibrate your system for temperature and give a sigh of relief if you don't need to.

ADCs and DACs are characterized by their sample frequencies (how fast they work) and number of bits. An 8-bit ADC has 256 levels in its digital signal; a 16-bit ADC has 65,536 levels. What you need depends on your application, but those two numbers can drive how your whole software system is organized by constraining the speed of your processor and the amount of RAM you need. How fast your system can process data is called its *throughput* or *bandwidth*.

Digital Sensors

If I have made you a little nervous with analog sensors and the arcane world of signal processing, I do apologize. However, you'll like digital sensors a lot more. Sometimes called a *smart sensor*, a digital sensor will do the analog to digital conversion for you, ideally giving you exactly the data you need. As with analog sensors, the range of digital sensors is staggering. Although digital sensors are usually a bit more expensive than their analog counterparts, the price is often worth it because they produce meaningful samples and are less susceptible to external noise (see “[EMI: Electromagnetic Interference](#)” on page 154).

As you look at the datasheet for the digital sensor, the primary concern is how to communicate with it. Does your processor have that type of interface? The next question pertains to throughput—can your software keep up with the amount of data the sensor is going to send? What is it going to do with the data? How long will it take?

EMI: Electromagnetic Interference

As signals travel along wires, they pick up noise from all the things around that have clocks. Wait a minute, you say—my system has a clock? Probably more than one, actually.

So you can't expect your analog system to be free of noise. The wires that the analog signal travels along are like antennas picking up static from everything around them. Your electrical engineer can shield the signal and reduce its susceptibility to noise (usually by putting a shiny metal box connected to ground, called a Faraday cage, around the signal).

Sometimes, before you can fix the issue, you need to find the culprit causing the noise. That can be difficult. If it is not external to your system, it could be any of your communication paths, any of your peripherals, or your processor itself. Even worse, it may be some combination of these interfering in tandem, giving you hard to reproduce errors. Radiating noise internally is not the only problem.

Sometimes, after bring up but before the end of your project, your electrical engineer will take your system off for EMC (electromagnetic compatibility) testing. If your system has a clock that is over 9kHz, in the US, it needs to go through EMC testing to ensure that it doesn't radiate signals that will interfere with communication (to comply with FCC part 15).

This test will put the unit into an anechoic chamber and monitor it at many radio frequencies to see if it emits any noise above the tolerated level. As you prepare the software used in hardware bring up, you may need to put together an EMC test version that will transmit data through all communication pathways as fast as possible to generate chatter and verify that nothing much radiates from your device.

The same EMC test lab usually can blast your unit with different frequencies to see if it is susceptible to noise in any particular range.



Although digital signals emit more noise to the environment, they are more immune to outside sources of noise.

Actuators

Actuators are what your software uses to make a mechanical impact on its environment. The simplest actuator is a *solenoid* which you can think of as a button in reverse. If you set it high, the solenoid is in one position. If you set it low, the solenoid is in another. These are useful for locking things or turning on and off valves. However, moving things is a lot more fun if you use a motor.

Motors

Whether you are making a bipedal robot or moving a widget along a conveyor belt, motors are where the systems really get to play in the real world. Unhappily, while motors are nifty and you should totally try them out, really great motor control requires its own book. I'll give you some things to look for but I can't give you anything more than an orientation to motors.



Just as you can buy an easier interface using a digital sensor, some motor assemblies include a motor controller chip that will implement an algorithm compatible with the style of motor.

First, what kind of motor are you looking at using? There are many types of motors and your application will drive what type you use based on a combination of charac-

teristics including price, life span, torque (turning power), energy efficiency, and power consumption.

A *stepper motor* is a popular choice though they can be relatively expensive. A stepper motor can be positioned precisely and will stay there (high holding torque). These are the simplest motors to use because you can run them open loop (without any feedback mechanism). Because they have a number of magnets to turn the motor shaft, stepper motors require multiple I/O lines. Each I/O line gets energized in order to cause the shaft to step forward to next position (this can cause a bit of a jerk). When you want the motor to go smoothly in one direction, you need to make the I/O lines dance in a complicated pattern (dictated by its datasheet).

At the opposite end of the motor spectrum is a (brushed) *DC motor*. These are very cheap and usually only require one I/O line (well, two if you want it to be able to go backwards). The speed of motor depends on the voltage applied. You can use a PWM to give you more control than off and running full speed. (We saw PWM control in [Chapter 4](#).) Motors don't usually like to be jerked on and off in a digital manner so using PWM to give smoother velocity profile that will increase your motor's lifespan.

The torque of a brushed DC motor depends on the current so your I/O lines will probably go through some additional circuitry to get more oomph (your processor probably only sources enough current to drive a very wimpy DC motor). A difficulty with DC motors can be stopping them at the precise position you want; they often require a feedback control system.

Because it doesn't have the jerkiness of the stepper and it doesn't have any parts (brushes) that touch, a *brushless motor* usually has a longer life span. As with a stepper motor, they need a few I/O lines working together to achieve rotation. As with a DC, it is easier to control the speed than the position. They are power efficient and usually priced between stepper and a brushed DC motors of similar characteristics. Also, just to make things a little confusing, a brushless motor is generally DC (though it can be AC).

Position encoding. The second most important thing to understand about your motor system is: how will you know when it gets to wherever it is going? With a stepper motor, you can count the steps. Even so, when the system boots up, how will it know? Most motors (stepper or not), have a home position that puts the motor in a safe position and tells the software the position. Usually home is at an extreme of the motor travel direction (e.g. all the way to the left or at the bottom). Finding home is a matter of moving the motor slowly in that direction until the home position is sensed. Generally, home is the zero position of the system and position counts from there (though sometimes home is the center of travel instead of one edge).

In a stepper motor, the position is counted based on motor counts. In any other type of motor, you'll need another type of sensor called an *encoder*. Often optical, it is usually a series of black and white (or reflective) areas that let your software treat the signal as a digital input. A rotary encoder measures the angle of the motor shaft and is usually

absolute (3 bits will let you know which of 8 positions your motor is in). A linear encoder measures the position of the motor along a straight line and is usually incremental so the software must keep track of where it is (in reference to a home position).

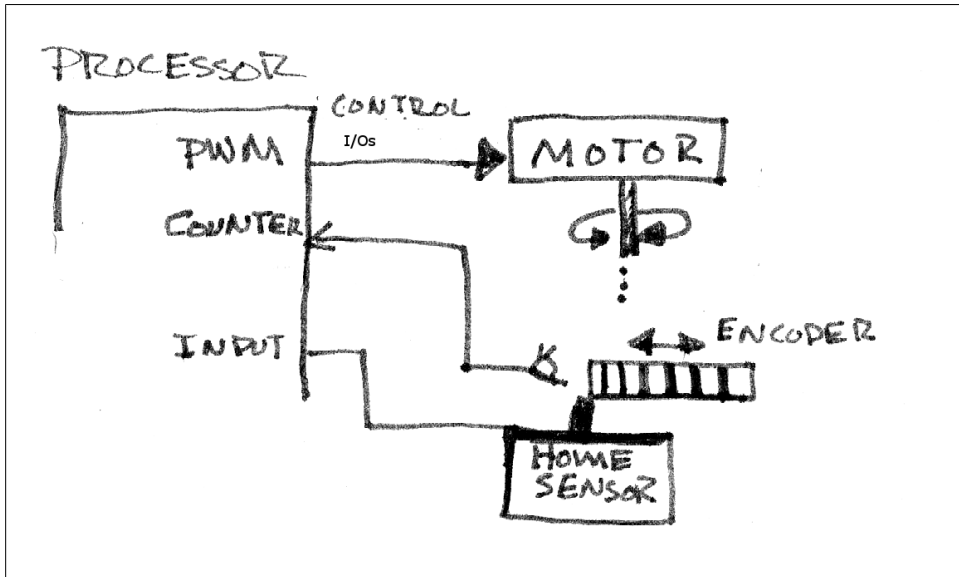


Figure 6-3. Motor encoder system

Figure 6-3 shows a simple motor system with some PWM control lines going to a motor. As the motor rotates, the encoder moves back and forth (the dots indicate the mechanical black box that includes gears and whatnot). A sensor sees the black and white areas on the encoder. The sensor connects to a counter on the processor. Initially, the motor goes through its homing routine. Once there the software can count how many lines it has on the encoder. Your processor probably has a feature that will do this sort of counting, check in the Timer/Counter section of your user manual.

PID Control

Controlling the motor is not as simple as telling it to go until you get to a position and then telling it to stop. By then, the motor has overshoot the position. You could tell it to go back but then you will overshoot again. Unless you are trying to rock a baby, you don't want to keep going back and forth like that.

The most common way to handle this problem is to use PID control. PID stands for proportional, integral, and derivative. Each term handles a part of the problem and together they can tell you how much power to send to the motor (usually in terms of PWM level). It starts with an error term that is simply the difference in position between where the motor currently is (*process value*, PV) and where you want it to be (*set point*, SP).

The proportional term is easy to understand. It is just a constant multiplied by that position error.

```
error = goalPosition - currentPosition;  
PID.proportionalTerm = PID.proportionalConstant * error;
```

So when the motor is far from the goal, your software gives it a lot of juice and when you are close to the goal, it powers down. Sometimes this is all you need. However, starting the motor full-on when you are far away from the target will cause it to jerk which is bad for it. Further, you may still overshoot the target leading to oscillations around the goal position. Or if your error is small, the proportional term will never give the motor enough power to actually move so you end up stuck in not quite the right position (steady state error).

The integral term helps these problems. The integral term adds the error over time:

```
PID.integralSum += error;  
PID.integralTerm = PID.integralConstant * PID.integralSum;
```

Not only is the integral term proportional to the amount of error (as the proportional term is), it takes into account how long there has been error. After the motor has been commanded to move, the error has only been around for a short time. But after a few cycles of the PID control loop, if the motor hasn't gotten close to its destination, the integral term lends oomph, accelerating toward the goal. However, it is kind of like momentum. Once the integral term gets going, it tends to cause overshoot, even worse than the proportional term would alone. However, the controller will find its way back and instead of constantly oscillating around the set point or stopping with some error, the integral term will cause the system to settle in the right place.

The derivative term reduces the overshoot caused by the integral term:

```
PID.derivativeTerm = PID.derivativeConstant * (error -  
previousError);
```

This makes the controller decrease the output if the error is decreasing. This adds some friction to balance the proportional and integral terms, usually counteracting the PI terms as the error term gets smaller.

The output of the PID controller is just the addition of each of the terms. [Figure 6-4](#) shows a reasonably well-tuned interaction between the terms and their PWM output as the system is commanded to go to position (or temperature 100). Note that with the integral and derivative terms, the output of the system feeds back into the controller so this is a feedback system. As such, it can be unstable. Too much proportional control can make the motor oscillate (or ring). Too much integral control and the motor may smash itself to bits as it overshoots the position. Too much noise in the error channel and the derivative term can lead to random results. Feedback systems that get out of control can do dangerous things.

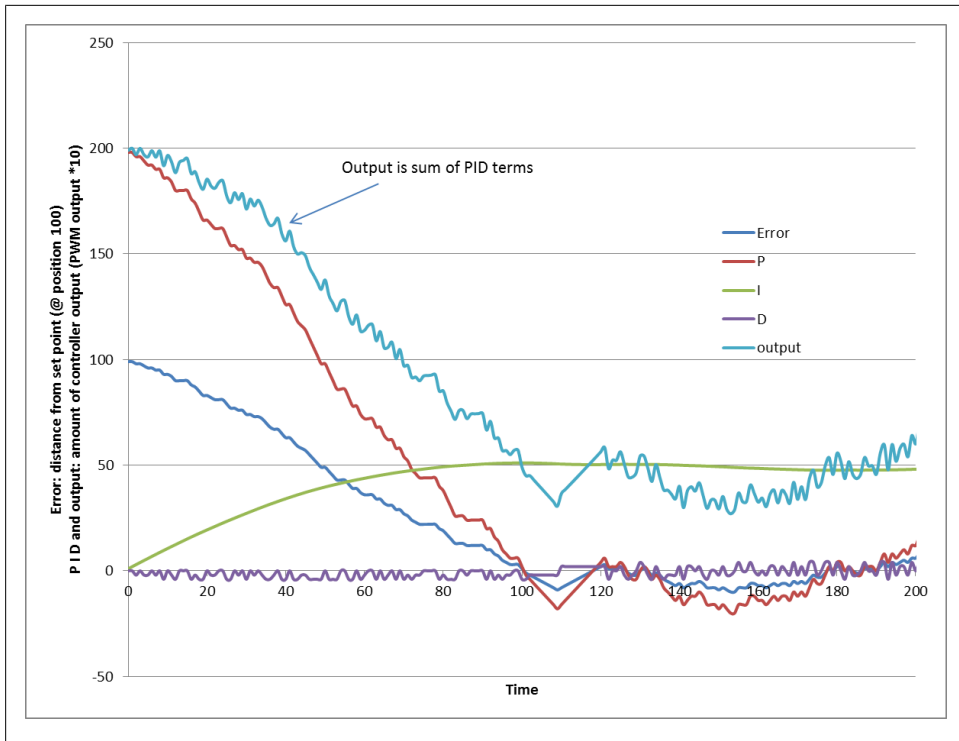


Figure 6-4. PID response over time

However, there are many, many engineering problems where a PID controller provides a good solution. I've used PIDs with motors, heaters and modeling spring systems. This control method is widely used and well understood (consider it an engineering design pattern). And you can see from the above code snippets that implementing one in software is simple.

However, those constants? The ones that describe how much weight each PID term brings to the output? Figuring those out is a painful process called *tuning*. There are some easy guidelines you'll find- start with proportional control until you get it working pretty well, then add integral until it works better, then add a little derivative to prevent too much overshoot. Three weeks after the project is supposed to end, you may still be cursing the person who gave you this advice telling you to just tweak the parameters until the motor works.

There are some mathematical tools to help you with tuning (e.g. Ziegler-Nichols method). Despite the mathematical tedium and having to estimate some inputs (the real-world doesn't match the mathematical ideal), I recommend using something other than trial and error. These methods often require a formal and deep understanding of the problem.

Note that the PID concept is pretty standard but implementations vary. For example, as the derivative term is sensitive to noise in the error signal, many implementations will use an average of error over several cycles. This hardens the derivative term against noise caused insanity but makes it less responsive to the system. There are many dozens of these little tweaks to ensure better performance and deal with system-specific issues.



At this point, I have given you enough information to be dangerous but not enough to do a good job. “[Further Reading](#)” on [page 195](#) suggests some books to help you with implementing and tuning a PID controller (and some other control methodologies).

Displays

Your system output may not only be in the realm of actuators. Some embedded systems have displays for dealing with those pesky humans. As with sensors, your display may be analog or digital, as simple as an LED or as complex as an LCD with touchscreen. The range is huge.

Segmented Displays

We've already spent enough time on individual LEDs in [Chapter 4](#), even making them dim using PWM control. A seven segment display is probably the next most basic display that you'll see. Most seven segment displays are ordered as shown in [Figure 6-5](#).

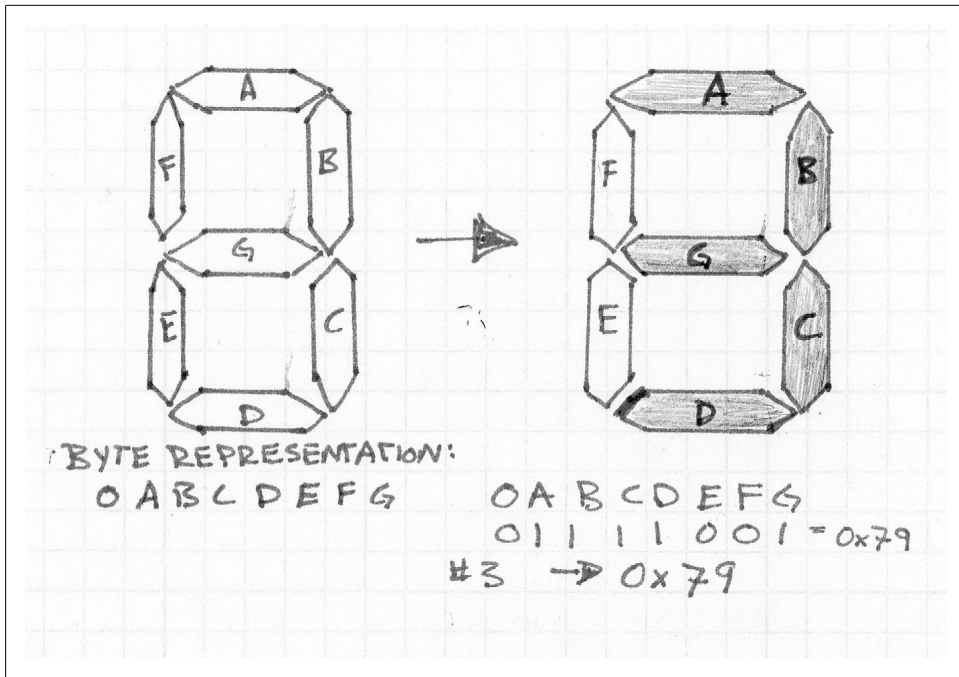


Figure 6-5. Seven segment display

Each LED in the segment can be represented as a bit in a byte (the extra one bit is sometimes used for the decimal place segment found on some displays). The segments are accessed in order: *abcdefg* (or sometimes *gfedcba*). So if you want to put a 3 on the display, you need to turn on the LEDs at *abcdg* and turn off the LEDs at *fe*. In hex, using *abcdefg*, you'll want to write 0x79 to the display to represent the number 3.

The actual implementation here is not that important. The critical idea here is the decoupling between human readable data on the display and the representation in the code. Get used to having to map between the display's context and other parts of the software.

Ok, taking a step back, do you remember the solar powered calculators? The LCD display with eight digits (each one a seven segment display)? The ones that were amazing technology in the 1960s but were handed out for free by the 1980s? They also had at least 17 buttons. However, the processors that ran inside those calculators didn't have 81 I/O lines (17 (buttons) + 8 (places) * 7 (segments in each character) + 8 (decimal points)).

As noted in above, you can matrix the button inputs: 17 buttons can use 9 lines (4x5) and you can get a three more buttons for "free". The same method can be used to matrix the display outputs. Using row/column scanning, for 8 characters with decimals (64 segments), we can use an 8x8 matrix so only 16 I/O lines are needed. With buttons,

the response time depends on the number of columns because you cut time spent looking at each column into that many slices. With outputs, the time slice is how much time each segment can spend on. So, if you do an 8x8 LED array, the LEDs will only be on 1/8 of the time. Most LEDs are so bright that this doesn't matter. With LCD (such as those found in our solar calculator), the limited amount of on-time can make segments look washed out and difficult to see, particularly when the power is low.



LCD segments don't run with a constant “on” like LEDs do. While they can be matrix'd, it is more difficult because need an AC signal. You'll usually use an LCD controller but if you want to try it (or just understand it better), check out this [ST Microelectronics AN1447 Application Note](#).

The 8x8 matrix is very convenient because it lines up with our character mapping very well. We can either power all of one character at one time or we can power all of the a-segments, then all of the b-segments and so on. Either method works. As long as you refresh the display fast enough (get through all of the segments faster than 30Hz), the user will never know.

Pixel displays

However, seven segments per character isn't enough to represent much. You can choose a fourteen segment display to get decent looking letters for English application (or sixteen segment display to incorporate Latin letters and Arabic digits). At some point, you might as well use a dot matrix display and build your digits, characters and graphics out of dots. Well, dots sound so plebeian, let's call them picture elements. Naw, that takes too long, let's call them *pixels*.

Now your mapping between the number 3 and what appears on your display may be even further apart than 3 and 0x79. Each thing you want to put on the screen needs a *bitmap* to describe what bits are on and what bits are off. Anything that goes on the screen is a *glyph* whether it comes from a bitmap or whether it is generated by the code (like a graph).

All of your bitmaps together are called your graphic *assets*. Graphic assets are usually split in to fonts and other graphics (pictures, logos, etc). As before, the goal of designing a display subsystem is to keep images of the data loosely coupled from the data itself. This gives you much greater flexibility to change screens, give your system a new look and reuse your display subsystem in a different product.

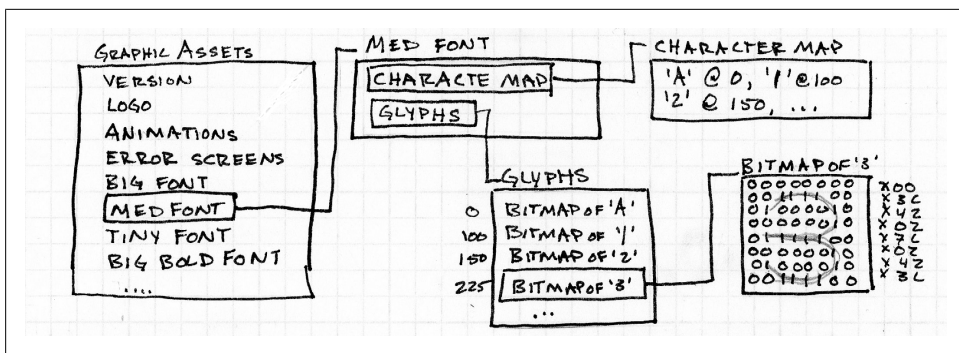


Figure 6-6. Graphic subsystem organization

Figure 6-6 shows one way to set up the graphic assets. First, there is a table of data, usually with a header file that tells you where the logo is or the offset to the animations. It is important to version the file (and the asset list); if they change without your software knowing you could display garbage to the screen.

To put the number 3 up on the screen, first you need to know what representation of 3 to display: Is the bitmap going to be small or fill the whole screen? Is it going to be bold or italicized? Each of these options will lead to a different font set in your assets. Once you know the font, you'll need to map from the character to the bitmap, usually by searching through the character map to get an offset into the bitmap glyphs.

Bitmaps can be stored in monochrome (1 bit per pixel) as shown in Figure 6-6. Look at all the wasted data though, that can be compressed. That may make getting the asset faster from wherever it is stored (often off chip flash). However, uncompressing the data becomes one more thing for the processor to do.

One form of compression that may not slow down the processor at all is *run length encoding* (RLE). Sections of data that are the same are stored as a single data value and a count rather than each pixel being replicated. This is great if you've got a lot of the same color on the screen. For example, you might get a stream of white pixels (W) and black pixels (B) that look like WWWBWWWW. This would encode to W4B2W4. If the data is stored as one bit per pixel (so the whole image is only black and white), that would give you no savings. However, if the image is one byte per pixel (or two or three bytes per pixel), RLE provides significant compression without much processor overhead).

Display subsystems end up pushing a lot of data from one place to another. A monochrome bitmap is fairly small. In the figure, the number 3 glyph will be 8 pixels wide and 8 pixels wide so it can be represented in 8 bytes. However, it will probably look terrible with those blocky edges. To make it look better, you'll want to add antialiasing which softens the edges by putting grayscale pixels in. Figure 6-7 shows a little bit of smoothing on the number 3 bitmap, it uses five levels so it would need three bits per pixel.

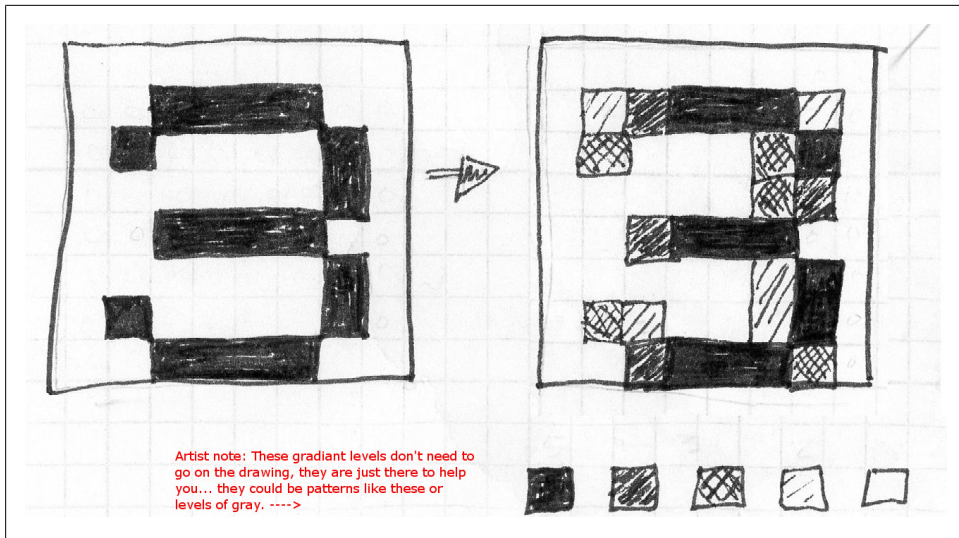


Figure 6-7. Antialiasing of a character

With anti-aliasing, the bitmap is no longer monochrome. It could be eight bits per pixel to describe the antialiasing which means the number 3 glyph takes up 64 bytes, still a really small amount but you're on a slippery slope. Before your whole A-Z, a-z, 0-9 monochrome font could fit into 500 bytes but now it is almost 4kbytes and that is for 8 pixel high letters. Once you've got a 2.4in LCD display (24 bit color, 240x320 pixels), each screen can take up to 225kbytes.

Coding a display subsystem often means optimizing your code so the parts that handle this data throughput don't cause problems with the screen. For example, *tearing* is when the screen refreshes showing half of the new image and half of the old image. Sometimes that is a synchronization issue but often it is matter of setting all of the pixels to their intended values as quickly as possible.

Note that how you pack the assets into the storage mechanism is important. You will need to make a trade off between having the assets take up a smaller amount of space or have a simpler interface for the processor. Another issue might intrude as well: what if your display has to go into the product enclosure upside-down due to electro-mechanical requirements? While it may require more front end processing to put the assets in a different orientation, it is better than having the processor turn them upside down and backwards each time the asset is read.



Being an embedded systems specialist does not excuse you from writing programs to pack data assets. Since you'll be the one using the information, you'll find your life is much easier if you control how the graphics are stored.

Finally, once you have the bitmap ready to send the screen, you'll need to know where to put it. [Figure 6-8](#) shows the number 3 glyph on the screen.

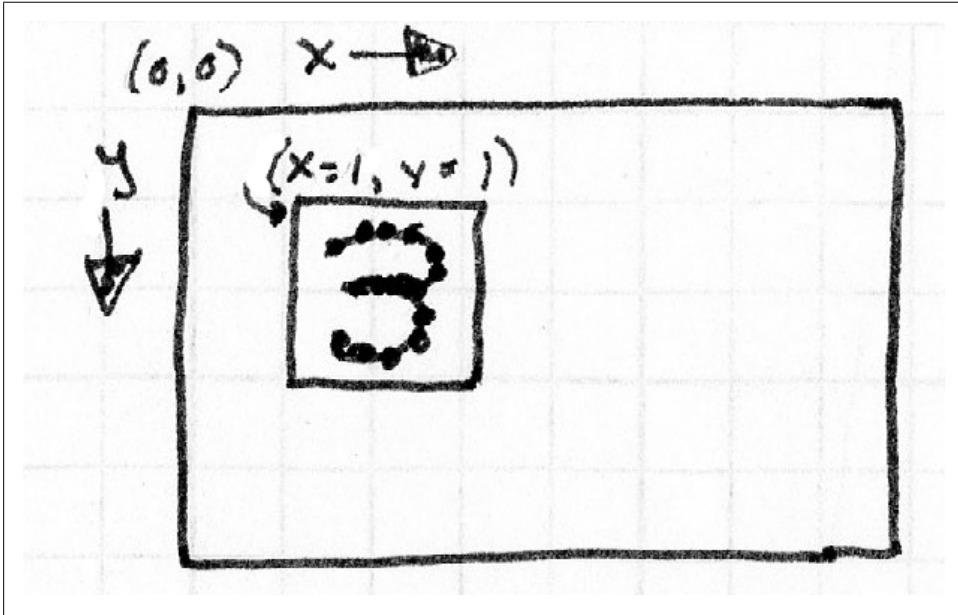


Figure 6-8. Number 3 on Display

To sum up the steps:

1. Determine what is going on the screen and where.
2. Look in the assets to find the correct font.
3. Find the character in the font.
4. Read the bitmap, possibly uncompressing it.
5. Send it to the display, ideally right after the last refresh.

To put up a logo or other fixed graphic, combine steps 2 and 3 into one: looking up the image glyph.

Hosts and Other Processors

From your processor's point of view, a host computer is just another peripheral to talk to. Sure, the host might change your processor's code or instruct it to move the motor to a different location or ask for the data you've collected. But all of those things are just digital communications. In fact, if you've been sending bytes to your display or your motor, it has a processor. And if your sensor is a digital one, it has a processor. Each processor is the center of its world so, to the digital sensor, your processor is a host.

Just as each of your peripherals has a documented interface, your system's interface to its host should be reasonably well documented (even if you control both sides). Putting a documented interface in place will let the two systems change independently. As for implementation, the command interpreter in [Chapter 3](#) is a good place to start.

As more of our everyday objects enter the network world, we build up detailed pictures of the environment. Whether it is a fridge that sends a shopping list to your phone when you've entered a grocery store or a scattering of small sensors monitoring a forest for fires, we are building *distributed sensor networks*. Increasing the intelligence of the sensors makes each one better able to deal with irregularities in its particular location, making the whole network more robust. Note that the information can return to a host for integration and processing (i.e. forest fire sensors) or the information can be generated with each piece contributing its knowledge in a host-less system (i.e. groceries).

The strength of the sensor network is that each element works together to construct information that is greater as whole. However, that is also their weakness; if the system cannot communicate, it is much less useful. Further, if enough sensors go offline, the system can become degraded to such a degree as to be useless.

So Many Ways of Communicating

Whether you are communicating as a host to a peripheral or as a sensor to a host, the important thing is the clock.

Sometimes the clock is explicit so one side generates it (usually the communication master). In other communication methods, the clock is implicit, agreed upon by both sides ahead of time (like knowing the baud rate of your serial device). The clock not only controls the speed of communication (throughput), it controls the existence of the interactions. Anytime you need to investigate a new communication method, the first question to ask is “where does the clock come from?”

OSI Model

The Open Systems Interconnection model (OSI model) is a way to talk about communication systems as a series of layers. Each layer provides some feature to the one above it to build up a communication pathway.

This model is far more complex than you'll need for most embedded systems but by having an idea of how a big complicated, communication pathway works, you'll get some ideas about how little ones work. Hint: most of them combine (or skip) layers.

Table 6-1. OSI Model

Layer	Function	Questions for this layer	On a PC's Ethernet
1. Physical	Provides electrical and physical specification	How many wires connect your processor to a peripheral? At what voltage? At what speed?	Ethernet cable
2. Data Link	Describes how bytes flow over the physical wires	Do the bytes have parity checking? How many bits are sent and received in each frame?	Ethernet (802.xx)
3. Network	Gets a variable length of information (packets) from one place to another	How is each system addressed? How to break up (and re-form) big blocks of data into amounts that can go over the communication pathway?	IP
4. Transport	Moves blocks of data in a reliable manner, even if those blocks are larger than the lower levels can handle	How do you count on data being received even when there is a glitch in wires? How are errors recovered from?	TCP
5. Session	Manages a connection between local and remote application	How to send this data from here to there?	Sockets
6. Presentation	Provides structure to the data, possibly encryption	How is the data organized when it is sent?	TLS and SSL
7. Application	Takes user interaction with the software and formulates a communication request	What command to send when a button is pressed?	HTTP

I've put [Table 6-1](#) upside down because the first few layers are the most important for embedded systems. While you may need to know how the data is sent to a flash chip or a received from a digital sensor, you won't get many options for implementing the presentation layer; that information will be specified in the datasheet.



Here are some mnemonics for learning the OSI layers, from the top down:

- All People Seem To Need Data Processing
- All Penguins Stand Too Near Deep Pools

Or from the bottom up:

- People Design Networks To Send Packets Accurately
- Please Do Not Touch Steve's Pet Alligator

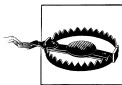
For an embedded system, most of your attention will be spent on the data link layer and how to move bytes from one place to another.

Ethernet and WiFi

Because it is so complex, using Ethernet usually implies an operating system. Since this book doesn't make any assumptions about you having an embedded OS, I'm not going to spend many words on this. However, as systems become more intelligent, Ethernet becomes more important. Due to the standard infrastructure around Ethernet, it is easily integrated into a large system (such as those distributed sensor networks). Finally, while Ethernet is the data link layer, it implies that there will be a network and transport layer involved so your software will interface to TCP or UDP. (TCP provides a reliable communication path so your data always gets there but it is more complicated than UDP.)

Communication reliability isn't that important if you are communicating with an LCD controller that shares board space with your processor. However, if your system is in the middle of the ocean monitoring for tsunamis, you may want something more reliable than serial communication.

If you are using a wireless network, you definitely want something reasonably reliable. Radio networks are hard. I mean, they are really hard. Murphy (of Murphy's Law) loves radio networks.



One of the most difficult parts of dealing with a radio network is their dependence on the environment. I know of one system that worked well on its install in November but gradually stopped working as spring came. Tree leaves absorb 2.4GHz radio waves.

There are amazing and fascinating applications of networked systems. However, when faced with the cost of a good radio, most companies flinch. At that point, someone will say, “let's just implement our own, the components aren't that expensive.” I know you want to agree as your product is amazing and the radio cost is a big hurdle to make it truly compelling in the market. I've been there, thinking “how hard could it be?” Take how hard you think it could possibly be and multiply by at least one order of magnitude.

Don't reinvent this wheel, as expensive as it seems. Spend the time making your system work. Once you've gotten that done, you can cost-reduce the product to implement your own Ethernet controller or wireless network. In the meantime, buy the radio off the shelf.

Serial

At its broadest definition, serial communication is the process of sending data one bit at a time over a communication pathway. Under this definition even Ethernet is a serial form of communication. Let's narrow it down a bit and say a serial port is something that can be implemented with a universal synchronous/asynchronous receiver/transmitter (USART). That probably doesn't help you much, does it? Let me define some terms and then we'll look at the most common serial communication methods.

There are UARTs (you-arts) which implement things like RS-232, the default protocol when someone wants to hook a *serial* cable to your computer.



Even high-end Internet routers often have a serial port for configuration or debugging. Not only is it cheap to implement, a serial port is like a torx screwdriver; most people don't know what they are. Using one is akin to admitting you think you know what you are doing.

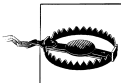
UART is missing the S in USART (use-art). The S is for synchronous. When a serial port is *synchronous*, both ends have to send bytes at the same time, always trading data even when one of them doesn't have anything important to say. SPI (Serial Peripheral Interface) is a popular bus that is synchronous.

Another important factor when considering serial ports is whether you have two wires to communicate upon (transmit and receive) or only one wire (which switches direction depending on whose turn it is to talk). Having two wires is called *full duplex*; sharing a single wire is called *half duplex*. 1-wire bus and I2C are two of the most popular half duplex protocols. Of course, half duplex serial ports are a little more complicated to debug because you can't be sure on your oscilloscope (or logic analyzer) which chip is the source of the bytes.

When encountering a new peripheral bus, the things to understand include:

- How is the clock generated? Is it implicit (all parties agree upon it ahead of time) or explicit? If it is explicit, does the clock generator (usually the bus master) have any other special responsibilities?
- How many hardware lines does it need?
- Synchronous or asynchronous?
- Full or half duplex?
- Is it point-to-point or is there addressing? Is it in the protocol or done via chip select?
- What is the maximum throughput of the system?
- How far can the signals travel?

The real goal here is to figure out how long it will take you to implement a driver using the communication method in question. The easiest driver to write is one you've already got working on another project.



Many processors come with example code, particularly for their communication drivers. Sometimes this code is excellent, sometimes it is only an example to get it running (but not working robustly). Be sure to review this code if you incorporate it into your project.

Serial Peripheral Bus Profiles

Moving on from the higher level terminology and general questions, let's look at how some popular buses behave. This is still just an introduction. In the end, you'll need to look at your datasheet to see what the peripheral needs and your processor manual to see how to provide it.

Timing diagrams are often very important to getting a peripheral working. As you bring up a driver for the first time, expect to spend some time setting an oscilloscope up and trying to recreate those diagrams (in order to debug your driver).

RS-232 and TTL

It is funny how the word serial is overloaded to mean many protocols. When we come to talk about the most common serial interface (the one between your system and your computer), it has several names. I already mentioned UART with is the part of your processor that implements this interface. However, it sends out TTL level serial signals (0~3V) and another chip converts them to RS-232 level signals $\pm 12V$.

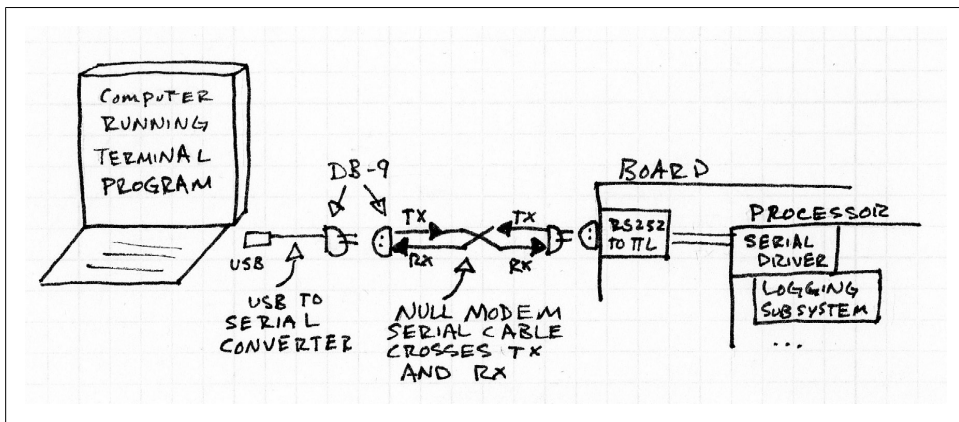


Figure 6-9. Serial connection from the embedded system to a PC

Figure 6-11 shows a possible debug pathway from your computer to your processor. While RS-232 defines eight signals (and ground), transmit (TX) and receive (RX) are the most important signals. You can interface your embedded system with your computer with those alone. Note that both systems have a TX and RX so you'll need to swap them to connect the TXs to RXs. This may be done in your cable (using a null modem cable) or on your board (in which case you need a standard or straight through serial cable).



Note that there are serial USB to TTL cables available from [Sparkfun](#) which can be used to take out the intermediate steps.

If you are hooking up to a cellular data modem or serial to Ethernet converter, you may also need some of the other RS-232 lines to indicate the modem is ready to receive data (RTS/CTS handshaking). At this point, even though the clock is implicit, you'll need to designate a bus master because the signals are not symmetric. In RS-232, the bus master is the DTE (data terminal equipment) and the slave is DCE (data communication equipment). Yes, pretty much every protocol has different names for similar things.

This form of serial resides at the physical and data link layers of the OSI model. There is no built in addressing scheme so it is only between two parties. Both parties must agree upon the baud rate (bits per second), parity, number of start and stop bits and the flow control (which can be hardware using RTS/CTS, software using Xon/Xoff or no flow control). The baud rates are usually between 2400 and 115200 (though it can go as high as 921600). Certain baud rates are very popular: 9600, 19200, 38400 and 115200. Because the data link layer require extra bits per byte sent (those start, stop and parity bits), most serial connections have a throughput in bytes about equal to their baud rate divided by ten. So if you've got a baud rate of 19200, your system is getting to get about 1920 bytes per second. Compare this with standard 100Mbit Ethernet (though 1 Gbit is pretty common too) which gives a throughput of around 11Mbytes per second, depending on the protocol. Serial is slow.

The relatively high voltage of RS-232 (12V!) means it can travel pretty far before the signals deteriorate, up to 50 feet (15 meters) with a normal serial (or null modem) cable. A special cable (low capacitance) can increase the distance by about a factor of 20.

RS-232 has been around seemingly forever and is continues to be widely used. Even though its demise is often heralded as new protocols take its place (Ethernet, USB, etc.), RS-232 keeps coming back because it is easy to implement in software and cheap to implement in hardware.

SPI

RS-232 works pretty well. But it has some deficiencies: about 20% of the bitstream is overhead; both sides have to know the set up (e.g. baud rate, parity); the RX/TX crossing thing is pain because it depends on your point of view which is wire is RX (or TX); etc.

So there is SPI (pronounced spy or S-P-I). There are four wires that connect your processor to a SPI peripheral:

Master-In Slave-Out (MISO) also known as Slave Data Out (SDO)

Receive for the master

Master-Out Slave-In (MOSI), sometimes known as Slave Data In (SDI)

Transmit for the master

Clock (CLK or SCLK)

Generated by the master, it is the clock for both sides of communication

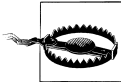
Chip Select (CS) or Slave Select (SS)

Generated by the master and one line per peripheral, it is usually active low

As noted above, SPI is a synchronous protocol so the master and slave both have to send data when the clock is going (0xFF is the traditional byte to send when you just want to clock out data from the slave). Since the master provides the clock, the SPI doesn't require that either the processor or the peripheral have a precision oscillator.

The clock speed can be slow (tens of kHz) or fast (up to 100MHz). The clock speed is the baud rate and there isn't any overhead so a clock of 8MHz leads to 1Mbytes/second, much faster than RS-232 and comparable to Ethernet at high clock speeds. Finally, the clock speed can be whatever your processor can clock and the particular peripheral can receive. There is no need to try to run at precisely 8MHz, it is acceptable to run at 8.1234MHz or 7.9876MHz. The flip side of running a clock so fast is that it doesn't travel very far, SPI signals don't usually leave the board (in a cable it travels less than a meter before deteriorating).

The protocol defines a chip select so multiple peripheral can often share a single SPI bus as long as there is one CS per peripheral.



Check your peripheral datasheets, not all peripheral play nicely on a shared bus.

A SPI driver is even easier to implement than RS-232 driver. The only tricky part is to figure out whether you want the data on the line to be valid when the clock edge rises or falls (clock polarity). Your peripheral datasheet will tell you what it wants and the processor manual will give you some registers to configure it.

The SPI interface is so easy that it is often implemented as a *bit-bang* interface. Sometimes your processor doesn't have the hardware interface to implement SPI. So you take four unused GPIO lines and make your own SPI interface. A timer becomes the clock and at each timer tick, SCK toggles and either an output bit is written to the MOSI line or an input bit is read from the MISO line.



You can create a bit-bang driver for any of the serial interfaces but it consumes far more processing power than a built in processor interface.

I2C

I2C stands for inter-integrated circuit and is pronounced eye-squared-see or eye-two-see. Like SPI, the bus has a master which provides the clock and starts the interaction. However, unlike SPI, the I2C bus master can change, allowing a peripheral to control the interaction. If SPI is a simplification of RS-232, I2C is what happens when folks start wondering how many wires they really need to connect a whole bunch of things together. Apparently, you need three connections: SCL provides the clock, SDA which provides the data (and goes both ways, making this a half-duplex protocol) and ground.

However, the simplicity of hardware design is paid for in software complexity. Not only that, because this protocol allows multiple peripherals (and multiple masters), it specifies an address scheme moving it beyond OSI layers 1 and 2 to layer 3 (network). On the other hand, I2C is fairly widely used so you should be able to find some example code to help you implement your driver.

I2C drivers invariably include a state machine to deal with the complexity of switching the direction of the bus between the master transmitting and the slave transmitting. The master starts communication by sending a 7-bit address and whether it wants to read from or write to the slave. The slave with that address then sends an acknowledgement (ACK). Next, the master sends a command to (or reads from) the slave and the interaction proceeds. When the communication is complete, the master sends a stop bit.

The number of components attached to the bus is limited by the address space (and by the capacitance of the bus). The 7-bit addresses are assigned by NXP. Many components let you set the last few bits of the address using pull-ups so you can put several of the same part on the bus. Alternatively, some components have slightly different part numbers that result in different I2C addresses).

Common I²C bus speeds are the 100 kbit/s (standard mode) and 10 kbit/s (low-speed mode). However, like SPI, since the master sends the clock, these frequencies don't need to be precise. Some peripherals implement faster speeds: 400 kbits/s, 1 Mbit/s and even 3.4Mbit/s.

While there is overhead built into the I2C bus, if you are moving large amounts of data around (i.e. reading from an ADC or an EEPROM), the bus arbitration and start/stop bits become a negligible percentage of the traffic. So with standard mode you can see 12.5 kbytes/s throughput on the I2C bus.

I2C isn't generally used to communicate to distant peripherals but it can go a few meters before the cable degrades the signal too much. As the protocol needs only two wires to go to many peripherals, your cable can be only four wires in total (the peripherals usually also need power and ground).



I2C is sometimes called TWI for two wire interface.

1-Wire

I2C isn't as few wires as possible; there is still the well-named 1-wire bus. It provides low-speed data communication and power over a single wire. (Well, you need ground too.) It is similar in concept to I2C (master, slave, arbitration, a software driver with a state machine) but it has an implicit clock of 100kbits/s. While this is a fairly low data rate, it can be used for longer ranges up to 10m (100m with special cables).

Like I2C, the bus can be used with a wide variety of peripherals including memory, sensors, ADCs and DACs. However, the most common 1-wire implementations are in authentication chips. These are used to authenticate the origin of pieces of the system which can be replaced. For example, if you are making a printer cartridge (or any form of consumable) and you want to deter competitors from making a drop-in replacement, a 1-wire secure authentication chip will force your consumable to send the correct password to your firmware before it can be used.

USB

All of the other serial protocols mentioned here can be bit-banged if you don't have the hardware (and you have enough processing power). I don't know that I'd try that with USB, it is pretty complicated. It is an asynchronous, full duplex system, with an implicit clock and up to 127 devices. And like most of the other protocols discussed here, it is *asymmetric* (it has a master that controls communication).

USB not only provides communication, it also provides power (5V, 500-900mA depending on version). It is fast (1.5, 12, 480, or 4000 Mbit/s depending on version). The cable can be of medium length (5 meters) and still maintain signal integrity.

Like Ethernet, the USB interaction is complicated enough to usually warrant an operating system (or a USB stack that goes far beyond a simple driver). Further, USB applications usually implement all or most of the OSI layers.

Miscellaneous Other Protocols

By now, I hope you are seeing the commonality of the different serial protocols. There is no need to memorize what is what. When you get a datasheet for a peripheral, it will tell you what it needs. And your processor will tell you how to implement that (or if it doesn't, Wikipedia will probably give you pseudo code for a bit-bang driver). The goal here is to think about what makes a protocol easy or hard to implement. Some take aways:

- The more OSI layers you need, the more complicated it is. Many protocols don't fit neatly in the OSI model but it is still useful to know the framework.
- Point-to-point is easier than a network which requires addresses.
- Half duplex is harder than full duplex because you have to switch the wire around
- Synchronous is easier than asynchronous because you know exactly when you'll get a byte.
- Having a master on the bus (asymmetric) is generally easier than trying to figure out who gets to talk when no one has control.
- Explicit clocks are easier than implicit.

So, say you need to implement a contact interface to a smart card. Smart cards use ISO-7816 which is a half-duplex, point-to-point, asymmetric protocol where the master provides the clock (in the 1Mhz-5Mhz range). How hard will it be to implement? With that information, it looks more complex than SPI but simpler than I2C. Add a little to your planning estimate because it is relatively rare and you probably won't be able to find good example code. Now, what if I tell you that it implements four or more of the OSI layers, though somewhat simplified? That should raise your implementation estimation by a factor of three (at least).

How about something more industrial like RS-485? Designed to travel long distances, it can be full or half duplex. When it is full-duplex, it is asynchronous. The clock is implicit (100kHz – 10Mhz) and only the OSI physical and data link layers are specified. There is usually bus master but that isn't required. With only that information, it sounds like a cross between RS-232 and I2C.

If you can find the commonalities between different protocols, then, once you've written something similar, you'll know what to expect when doing the implementation of something new.

ASCII Characters

If your test code for a new serial driver sends “Hello” as its first word, you can't expect to see an electronic hand waving on the oscilloscope during debugging. The letters get translated into numbers, usually 8-bit numbers according to the ASCII character encoding scheme.

Letter;	H	e	l	l	o
ASCII:	0x48	0x65	0x6C	0x6C	0x6F

So when you see “Hello” starting on your transmit line, it will start with ‘H’ which is 0x48 which look like: 01001000. In [Figure 6-10](#), I've shown how to make it a little easier to know where you are in the data stream by starting the transmission high to show the falling edge of the data. You might be able to see the start of the signal by triggering on chip select (if your protocol has one).

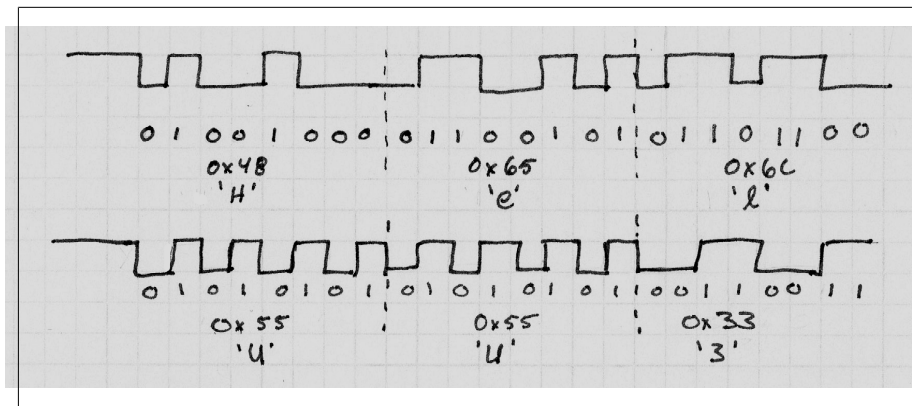


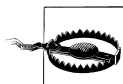
Figure 6-10. Compare ASCII Hello with UU3

Or you can select some letters that will be easier to pick out from on the oscilloscope. I like “UU3”. Each ‘U’ is 0x55 which means every other bit is set so it looks like a clock signal. And ‘3’ is serendipitously 0x33: 00110011 so it looks like a clock at half the speed.

Learning the ASCII chart is a lot like learning Morse code. It is very handy if you use it but if you are just learning for fun, don't expect long term retention. On the other hand, if you end up writing about one serial driver a year, you may start remembering the highlights. The ASCII chart is set up in a logical way so it is easier to remember only the important stuff:

'0' -> 0x30
'A' -> 0x41
'a' -> 0x61

The numbers and letters increment from those starting points (hence 0x33). You won't know where the punctuation and the control characters are but that summarizes the table well enough to talk to another embedded engineer in case you get trapped somewhere with binary communication.



While ASCII is the most prevalent encoding scheme in embedded systems, it can't fit characters from other languages into 8-bit words. Unicode uses 16 bits (sometimes more). If your product will be internationalized, figure out if Unicode is the right choice for you.

Parallel

Parallel is the opposite of serial. Where in serial, you send all of the bits over one line (maybe one line for transmit and one for receive if you have full duplex), in a parallel bus, you have one line per bit. It can take up many I/O lines. That tends to make a chip more expensive (I/O lines take up space in the silicon, the size of the silicon is propor-

tional to the cost to make the chip). Figure 6-11 shows a comparison in the number of lines needed for some of the most popular buses and common peripherals.

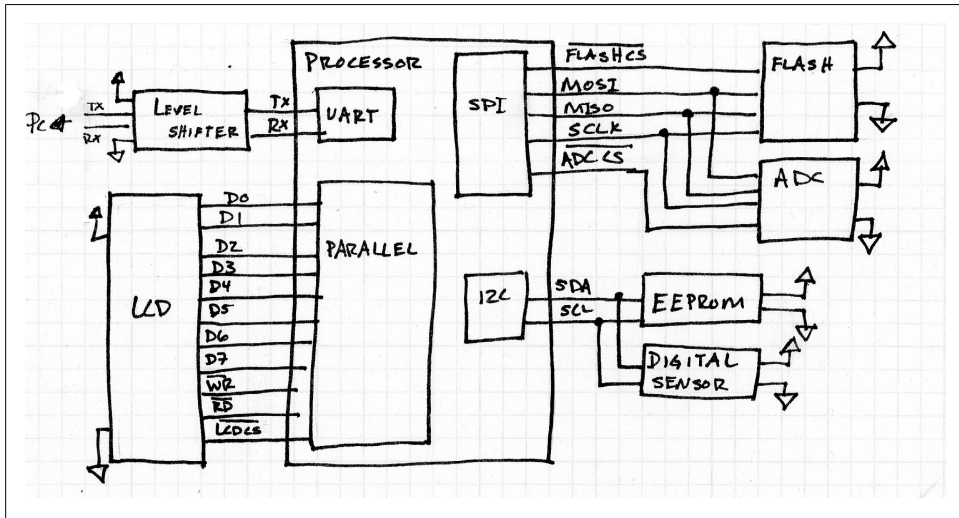


Figure 6-11. Comparing peripheral communication methods

By putting eight or sixteen bits on the bus at once, you can communicate very quickly. Almost always the data bits of the parallel bus are put together on a single bank of I/O pins. This let you set all of the bits by writing the data byte to the I/O register. Instead of setting a single bit as shown in Chapter 4:

```
LPC_GPIO1->DATA |= (1<<2); // IO1_2 high
```

You can modify several I/O lines at once:

```
LPC_GPIO1->DATA |= data & 0xFF; // put 8 bits of data on the parallel bus: IO1_0 to IO_7
```

This interface tends to be reserved for things that have high data throughput requirements: LCDs and external memory. As noted earlier in this chapter, a cell phone sized LCD (240x320 pixel display) needs 225kbytes per screen (if you update the whole screen).

In a serial bus that would mean 1.8Mbits per screen (225k*8 bits/byte). The update rate is at least 30Hz so you'd need to move 54Mbits/s, faster than SPI could manage. However with an 8-bit wide parallel bus, the bit rate per line goes down to 6.75Mbits. The data gets spread over the whole bus; the more lines in parallel, the higher the throughput.

Parallel buses tend to be half duplex with control lines to indicate which direction the data is going (read and write) and whether the bus is connected to a chip (chip select). The control lines tend to also be the clocks for the system so a write interaction would look like:

1. Select the chip.
2. De-assert the write control line.
3. Set the parallel bus to the data intended.
4. Assert the write control line.
5. Go to 2 until all data has been transferred.

Check the timing of your peripheral datasheet to figure out if you need some delay between them. All in all, implementing a parallel bus is pretty easy, on the order of a SPI driver though debugging is a bit harder because there are so many lines to look at.

Putting Peripherals and Communication Together

So we've seen the data generators and the data consumers. And we've seen how to send and receive data. However, there are some pieces missing between these two. Understanding how to move the data around in your software is critical to making a great system. Let's start with the big picture of the handling the data at a system level and then work down into the mechanics.

Data Handling

In [Chapter 5](#), we looked at systems that were event driven: things happen because sensors were activated, causing events which caused changes the state of the system. However, not all embedded systems can (or should) be set up as event loops and state machines.

There is a class of problems where the goal is to get data, process it, do something with the results and repeat. In such *data driven systems*, there are no events, just an ever increasing mountain of data for your software to process. Ideally, the system can wade through the data just a tiny bit faster than it is generated. Some examples of test driven systems:

- An airplane's black box continuously records audio and telemetry.
- A gunshot location sensor listens to its environment. Upon identifying an impulsive sound, it generates an event to send to a host.
- A reconnaissance satellite records image data, compresses it, and (when in range of a link point) sends it to Earth.
- An MP3 player reads data from its audio store, uncompresses it and sends bits out a DAC to generate a signal that sounds like music to your ears.
- A robot on an assembly line, shifting widgets from one conveyor belt to another.

A data driven system can be understood by looking at the flow of data. The rate of the data and how quickly it needs to be processed are the primary features of the system.

Reboots and errors can cause the system to fall behind. This is one type of system where failing gracefully is important. If the system had an error that caused it to fall behind, should it skip some of the data and catch up? Or should it do some sort of reduced processing until the system returns to normal? What are the consequences of missing data?

Happily, the implementing a data driven system is pretty straightforward because the processing on the data is repetitive. Consider the analog to digital system described in [Figure 6-2](#) where digital data comes from the ADC, gets attenuated and goes out the DAC. The system can usually be divided into a producer of data (the ADC) and a consumer of data (the DAC). These must remain in zen-like balance. As long as the processor can keep up with the ADC by running the algorithm and sending the data to the DAC, the system can run forever.

Most systems have some elements of an event driven system and a data driven system. As you consider your system, try to figure out what aspects belong to each. This will unravel some of the complexity of your software by allowing you to implement them separately.

Circular Buffers

The circular buffer is a key implementation tactic among data driven systems. Unlike stacks mentioned in [Chapter 5](#) where the last datum in is the first one out (LIFO), circular buffers are first in, first out (FIFO).

A producer puts data in the circular buffer at some rate. The consumer takes it out of the buffer, at the same *average* rate. However, the consumer can take it out in chunks while the producer puts it in dribbles. This lets the processor do other things while the data accumulates to the size needed for the algorithm. (Or it could go the other way, with the producer putting a large chunk of data into the circular buffer to be read out at a steady rate.)

A circular buffer needs to keep track of a chunk of memory, its length, where to put the next element that comes in (**write** or **start**) and where to get the next element (**read** or **end**). See [Figure 6-12](#). Note that understanding a circular buffer is easier when the read pointer is before the write pointer. This is true for everyone; most of the common circular buffer bugs happen when the read and write pointers are crossed even though the software will spend half of its existence like that.

When the buffer is empty, the read pointer is equal to the write pointer. This is easy enough to check. However, the problem is that when the buffer is full, the read pointer is also equal to the write pointer.

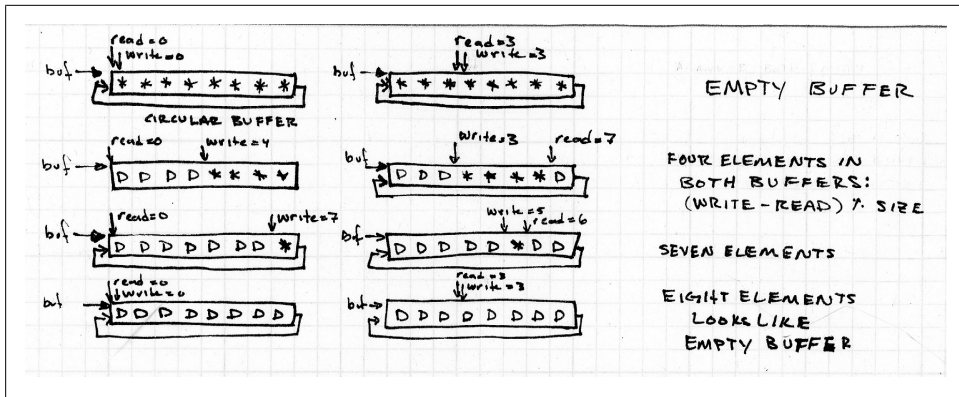


Figure 6-12. Example of pointers in a circular buffer

A common work around is to call the buffer full when the write pointer is one away from the read pointer. This is easy to understand and implement though it wastes an element's worth of memory. Another possible solution is to add a variable to keep track of the length by having an enqueue function increment the length and a dequeue function decrement it. Now you've traded a variable for a position in your circular buffer. Depending on the size of the elements in the buffer, that may be worthwhile.

Usually, one end of the circular buffer is an interrupt (either the producer or the consumer) and the other end is usually not. We saw in [Chapter 5](#) that all sorts of interesting things occur when you have interrupt code and non-interrupt code trying to modify the same variable. If you use the blank-element to know when your buffer is full, you can isolate the producer's variables from the consumer's, making the circular buffer interrupt safe as long as the read and write pointers update atomically. (Otherwise a request for the length of available data will give an invalid response.) However, since the pointers wrap around when they get to the end of the buffer, an atomic operation may seem unlikely.

Most common implementations in embedded systems constrain the circular buffer length to be a power of two to avoid this issue. So let's start with an implementation using a structure like this:

```
struct sCircularBuffer {
    tElement *buf;    // block of memory
    uint16_t size;    // must be a power of two
    uint16_t read;     // holds current read position: 0 to (size-1)
    uint16_t write;    // holds current write position: 0 to (size-1)
};
```

In the initialization function, we'll set the buffer pointer to a block of memory (one that holds a power-of-two number of elements). We will also need to set the size to the length of memory and the read and write offsets to zero. From there, we only need to implement a few other functions. The first is probably the most difficult to follow. It

does a wrapped subtraction to determine the length of the data available in the circular buffer:

```
uint16_t CBLengthData(struct sCircularBuffer *cb)
{
    /*
        *****
        |-----|-----|-----|
        write      read

    */

    int32_t length = cb->write - cb->read;
    if (length > 0) return length;
    /*
        bbbbbb          aaaaaaa
        |-----|-----|-----|
        read      write

    */
    return (cb->size - cb->write) + /* aaaa */
           (cb->read);           /* bbbbb */
}
```

Actually, that doesn't use the constraint of the buffer size being a power of two. A more efficient implementation looks like:

```
uint16_t CBLengthData(struct sCircularBuffer *cb)
{
    return ((cb->write - cb->read) & (cb->size - 1));
}
```

Even though the read and write variables are unsigned, the length be correct thanks to that bitwise AND. [“Representing Signed Numbers” on page 181](#) explains why.

Representing Signed Numbers

You know about unsigned binary numbers (see [“An Introduction to Binary and Hexadecimal” on page 76](#) for a refresher). [“How many bits in that number?” on page 101](#) showed the highest number you could fit in an 8-bit or 16-bit variable, where signed numbers were always half as much as unsigned numbers: the sign takes up a bit. There are several ways to encode the sign in one bit, consider how we do it in decimal with a sign to the left of the value. One way to do it in binary is to have the sign be the leftmost bit (most significant bit) while the other bits encode the number:

```
snnn nnnn = <1-bit sign> <7-bit number>
0000 0000 = +0
0000 0001 = +1
...
0111 1111 = +127
1000 0000 = -0
1000 0001 = -1
1111 1111 = -127
```

Note that there is a 0 and -0. Many encodings have that. It is a little odd and a little inefficient.

However, that sign and value encoding isn't how your computer normally stores numbers because there is a way to make processing faster using a different encoding. The ingenious method is called *two's complement*. In this representation, the left most bit still holds the sign of the number. And a positive number is encoded just as you'd expect. However, a negative is encoded by inverting all the bits and then adding one to the result:

```
snnn nnnn = <1-bit sign> <7-bit number>
0000 0000 = 0
0000 0001 = +1
...
0111 1111 = +127
1000 0000 = -128
1000 0001 = -127
...
1111 1110 = -2
1111 1111 = -1
```

I find the “invert and add one” instructions difficult to remember. However, the neat part of two's complement is that when you add a number and its opposite in sign, the numbers obliterate each other:

```
0000 0001 = 1
1111 1111 = -1
-----
1 0000 0000 = 0 (and a carry bit)
```

Being able to always add numbers (and not perform a different operation for subtraction) makes this number encoding very powerful.

Remember that -1 is equivalent to the highest unsigned value. If you write code:

```
uint8_t value = -1; // set to highest value this unsigned type can hold
```

Your value will be 255 (0xFF). To figure out other unsigned numbers, subtract normally from -1:

```
-1 = 0xFF = 1111 1111
-2 = 0xFE = 1111 1110
-3 = 0xFD = 1111 1101
-4 = 0xFC = 1111 1100
...
```

Let's see how this encoding works on the modulo math from this chapter. For example, here are the steps to do the wrapped subtraction of `cb->write` (5), `cb->read` (6,) and `cb->size` (8):

```
(cb->write - cb->read) & (cb->size - 1)
= (    5      -    6    ) & (    8      - 1)
= (          -1        ) & (    7        )
= (    1111 1111      ) & (    0000 0111 )
=    0000 0111 = 7
```

The power-of-two length makes the bitwise AND eliminate the sign information, leaving us with the correct result.

An empty buffer would return a length of zero. A full buffer would have a length of `(cb->size-1)`. Keeping that in mind, writing the enqueue function is pretty simple:

```
enum eError CBWrite(struct sCircularBuffer *cb, tElement data)
{
    if (CBLengthData(cb) == (cb->size-1)) { return eErrorBufferFull;}
    cb->buf[cb->write] = data;
    cb->write = (cb->write + 1) & (cb->size - 1); // must be atomic
}
```

The modification of the write variable in the last line of code needs to be a single instruction. Well, only setting to the write variable need to be atomic; the preparation `((write+1)&(size-1))` can be multiple instructions. These variables are declared as `uint16_t`; if this code runs on a 16-bit processor, the setting of the write variable will be atomic (but for an 8-bit processor, it won't be; in a 32-bit processor, usually the `uint16_t` write instruction is a single operation but not always).

Also note that the circular buffer rejects data that would overflow the buffer. This goes back to the question of how you want your system to fail. Do you want to reject new data as this code does? Or do you want to drop some old data to make room for the new data? What is your strategy for falling behind? I'll stick with the error for now but you do have options to consider.

Taking things out of the buffer is similar but deals only with the read offset:

```
enum eError CBRead(struct sCircularBuffer *cb, tElement *data)
{
    if (CBLengthData(cb) == 0) { return eErrorBufferEmpty;}
    *data = cb->buf[cb->read];
    cb->read = (cb->read + 1) & (cb->size - 1);
}
```



If you don't constrain the buffer size to a power of two, you need to use modulo arithmetic to wrap the pointers. Modulo arithmetic (like division) is a relatively slow operation, not the sort of thing you want to do in an interrupt. [Chapter 9](#) talks more about what forms of math are fast and slow on an embedded processor.

Note that the data element is copied out of the buffer to another location. In general, copying memory is not a good use of your embedded system's limited processing cycles (the act of copying) or limited memory (two copies of the same information). Why don't you just leave it where it is?

You can add another pointer to your circular buffer to indicate when data is free for use by the write pointer. However, this is where buffers start to get complicated. This is also where you might want to pull out a paper and pencil or get to a white board. For many circular buffer problems, the solution is far easier to see when you can draw out the options (see [Figure 6-13](#)).

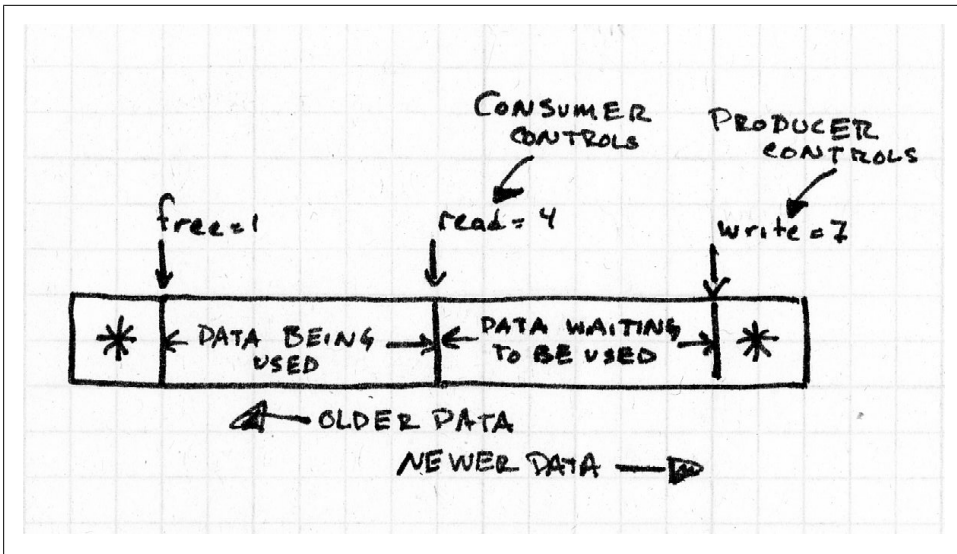


Figure 6-13. Circular buffer with multiple pointers

Once you get it straight in your head the code will be pretty simple. First, add a free variable to the structure (initialize it to zero). Next you'll probably need to know the lengths of the different sections of data:

How much is free and ready to be written

Wrapped subtract of write from free

How much data can be read

Wrapped subtract of read from write

How much data is currently checked out for reading

Wrapped subtract of free from read

Handling the free pointer is straightforward. When the code is done with the element it took, simply free it back to the circular buffer for the write pointer to use:

```
enum eError CBFree(struct sCircularBuffer *cb)
{
    if (CBLengthReadData(cb) == 0) { return eErrorBufferEmpty;}
    cb->free = (cb->free + 1) & (cb->size-1);
}
```

Note that element wasn't passed in because you have to free them in order.

You may want to do more than read parts of the buffer. Many data processing steps lend themselves to in-place modification. Once you get the hang of handling multiple pointers into the circular buffer, you may want to keep using them to avoid copies. Such streamlining will make your system faster.

Figure 6-14 shows the example from the analog sensors section. The analog data is sampled by the ADC. The digital data is put into a circular buffer (write) at a constant rate. The data may be obtained from the buffer at a different rate because the processing is done in chunks (you do something to several samples at once) or because the processor does something else while the data queues up. The processing module reads and modifies the data. It could put the data in another circular buffer for output to the DAC or it could return it to the same circular buffer (or modify it in place). Once the DAC outputs the processed data, the circular buffer element is free to be used again.

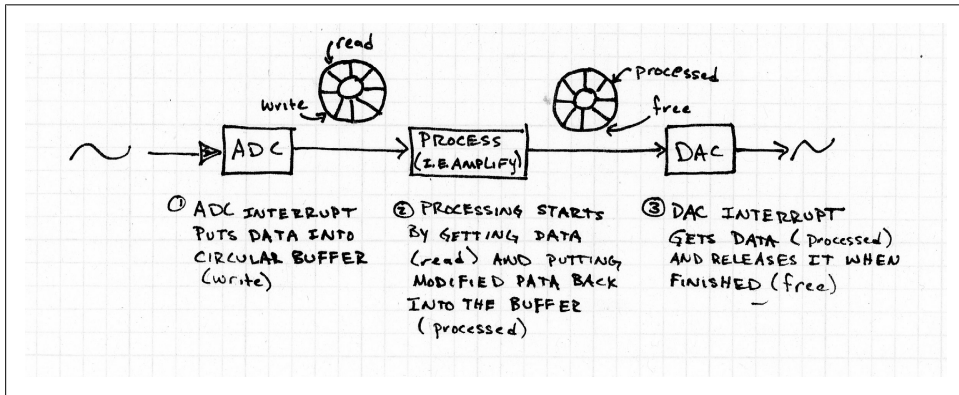


Figure 6-14. ADC to DAC data driven system with multiple pointers into a single circular buffer

To implement a system like that, you'd need to add one more pointer to the system called **processed**. As the pointers move independently of on another, adding another is fairly straightforward. The code is nearly the same as we added for **CBRead** or **CBWrite**. Don't forget to add a new length function(s) too.

Hardware FIFOs

You don't always have to implement a circular buffer. Sometimes the hardware will do it for you with a FIFO buffer.

Using a FIFO is pretty simple. To send data, write to the transmit holding register (THR). Writing to this address sends data to the transmit FIFO. The data will get sent when it reaches the end of the FIFO queue and the transmitter is available. Unlike the circular buffer, there is no dealing with pointers, you write to the THR to fill the FIFO and the data eventually reaches the pins. While the peripheral is streaming out the data stored its FIFO, the processor can do other things.

It isn't just transmit, you usually get both transmit and receive FIFOs. To get data from the receive FIFO, check to make sure something is available (using a status register) and then read the receive buffer register (RBR) to get the oldest data in the FIFO.



The send and receive registers (THR and RBR) can be the same register, changing roles depending on whether you write to it or read from it.

Processor FIFOs use a set of flags to signal the software. The status flags generally include: full, empty, half full, half empty. (If you are thinking that the FIFO has optimist/pessimist issues, usually a receive buffer signals half-full while a transmit buffer signals half-empty.) These levels can be used to interrupt the processor.

Some processors will let you set specific levels at which to interrupt. For maximum throughput, determine how long it will take your processor to get to the interrupt (latency) and how much data can be sent in that amount of time. For example, suppose you want to keep a steady stream of data flowing on the SPI bus. If you are running SPI at 10MHz and it takes you a maximum of 3 microseconds to respond to an interrupt, you need to get an interrupt when the FIFO has 4 bytes remaining (10MHz clock -> 1.25 Mbytes/s -> 0.8 us/byte, 3 microsecond latency / 0.8 us/byte -> 3.75 bytes).

A transmit interrupt usually fills the FIFO with available data. A receive FIFO interrupt flushes all of the data, putting it where the software can use it (often in a circular buffer). The goal is to balance the trigger levels so that you maintain a constant stream of data with the fewest number of interrupts.

Your processor manual will describe how to configure your FIFOs in more detail. FIFOs are usually 8 or 16 bytes deep. But what if your FIFO depth wasn't so limited? What if your FIFO was more like your circular buffer, using as much RAM as you need?

DMA

A processor that supports DMA (Direct Memory Access) can pass far more data than one that implements only a FIFO. To use DMA, you give the processor a pointer and a number of bytes to read. When receiving, the processor puts data from the peripheral until it reaches the byte count at which point it interrupts the software. Similarly, when transmitting, the processor moves data from the buffer to the peripheral, interrupting when the count is complete.

DMA is a lot like having another thread do the data handling for you. The configuration is relatively straightforward, look in your manual for the DMA registers: transmit pointer register (the buffer to write to), transmit count, receive pointer register and receive count. If your data comes in chunks (instead of a constant stream), you can use a DMA controller with one channel. If your data is constant stream, you may need to copy the data from the DMA controller to another location so your software has time to work with it.

To avoid the copy (which, as you recall, is an inefficient use of your processing cycles and RAM resources), many DMA implementations can alternate between two buffers.

This allows the software to use one buffer while the processor transfers peripheral data into (or out of) the other. This two buffer back and forth is called a *ping-pong buffer*.

DMA is used with peripherals that have very high throughput such as disk drive controllers and network cards. It is also used to in multi-core processors to communicate between cores. If you know how many bytes you'll be receiving (or you are transmitting a block of data and not just a dribble), DMA is a great way to reduce processor overhead, making your processor seem faster than it is.

Comparison of Buffering Schemes

How much does good data handling really matter? Well, let's compare how much processor time would be needed to implement an SPI interface, running at 1Mhz with a byte wide interface.

If your processor didn't support a SPI peripheral at all, you can implement a bus master via bit-banging. Set up a timer to interrupt at twice the clock speed (2MHz). On the every other interrupt, put the clock line down and set the outgoing data line (MOSI) value of the next bit in the transmit data byte (and shift the data byte). On the alternate interrupts, toggle the clock line up and read data on the incoming data line (MISO), setting the next bit in the receive data byte, shifting it to the next place. Every eight bits, you'll need to start a new byte. Overall, you don't have to do all that much for each interrupt, but you have a lot of interrupts and which means tons of context switching.

The next fastest implementation would be a hardware supported interface. When your processor datasheet says something like:

One Master/Slave Serial Peripheral Interface (SPI) – 8- to 16-bit Programmable Data Length, Four External Peripheral Chip Selects

Then you've got built in SPI. You can configure it to run at 1MHz (sometimes not precisely as this will depend on your processor clock and how you can use dividers, but usually you can get close enough). Then you'll need to configure it to interrupt when a byte has been exchanged, every 125 kHz. In that interrupt, you'll need to get the byte from the receive register and put it somewhere (e.g. a circular buffer). You'll also need to put the next byte in the transmit register. Not only is the interrupt shorter than the bit-bang version, there are many fewer of them.

Of course, if your FIFO is 16 bytes deep, you only have to interrupt at 7.8kHz. Well, you could if you interrupt when the FIFO is completely full. But that means your communication will fall behind if your latency is greater than 128us (1/7.8kHz). To give you more latitude, interrupt when it is half-full. That will still only give you an interrupt rate of 15.6kHz. At every interrupt, you need to move the bytes in the receive FIFO into another piece of RAM and move the transmit bytes into that FIFO.

Now for DMA. Well, if your goal is throughput, divide your RAM into a small piece for local variables and a large piece for a ping-pong buffer. If you have 32kbyte of RAM, you could save 2kbyte for variables and have ping-pong buffers that are 15kbyte each.

That means you'll get an interrupt 8 times a second, leaving you with plenty of time to actually do the processing.

Table 6-2 compares the processing difference of the interrupt overhead alone (assuming each interrupt has only 10 cycles of overhead, a fairly small amount). Instead of making a massive DMA buffer, I gave it a reasonable 512 bytes to work with. Note that to implement a bit-bang solution, you would need a processor that is more than 8000 times faster than a comparable DMA solution. That is just the overhead! A bit-bang driver is more complex than the DMA interrupt which just needs to switch pointers and set a flag for the processing to occur on the already full buffer.

Table 6-2. How much faster is DMA?

Buffering Methodology	Bit-bang interface	Hardware peripheral interface	16 byte FIFO (interrupt on half)	DMA with 512 byte buffer
Clocks per second taken up with interrupt context switching (10 cycle interrupt overhead)	20MHz	1.25MHz	156 kHz	2.44 kHz
Times faster than the bit-bang interface	1	16	128	8197

Adding Robustness to the Communication

No matter your throughput, you need to know that the data you received is the data that was transmitted. If you don't control both sides of the communication pathway, this can be tricky. Your SPI ADC may not have any facilities for ensuring data integrity. On the other hand, there is plenty that you can do when you do have some control.

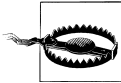
Version Everything Then Checksum It

In “Version Your Code” on page 24, I recommended that you have a version for your code and make it easy to get to from outside your system. Don't stop with versions there. Even though your product may be a self-contained widget on the outside, it probably has several components inside that can change independently. It is critical to version and checksum pieces of the system which can be attached and detached. It is less important to do it for components that are on the same board as the processor (though still pretty important if they get programmed separately). A version byte (or two) tells you when two parts of your system are incompatible. A checksum will tell you when components have been tampered with or have experienced hardware failures.

So, put a version in your flash memory so you know what assets are there. If you don't have enough code space to be backward compatible, at least you can give an appropriate error instead of putting random designs on the screen until the software crashes. The version can also act as a key to verify that external memories are programmed.

Running a checksum over a large flash is often prohibitively slow. However, if you checksum each asset as you obtain it from the flash, you can verify the asset went into memory correctly and the communication pathway didn't lose or mangle any bytes.

In the same spirit, add a version to your EEPROM in case the layout (or size) changes in the future. Right now, a larger EEPROM might be too expensive or unnecessary but things change. Give yourself the means to be flexible. Even if your EEPROM is small, adding another byte as a checksum will pay dividends when cosmic rays damage a chip.



While cosmic rays get jokingly blamed for many irreproducible errors in hardware and software, they really do happen. And they really do cause errors such as corrupted data in memory devices. *Radiation hardening* your system creates more work for each engineering discipline: mechanical (shielding), electrical (choosing components) and software (redundant pathways and error checking).

If you have control over both sides of a communication protocol (for example, if your company makes both an embedded system and a host for it to talk to), adding a version to the communication protocol will allow the pieces to grow independently.

I suspect you get the idea with versions and checksums but there is one more: version your hardware. How? Take three of your I/O lines and send them to a set of pull-down resistors. As you configure the lines, make them inputs with weak pull-ups. Next read the value to get a version. This lets your hardware describe itself to the software, possibly letting you reconfigure for different I/O configurations as described in [Chapter 4](#), as long as you don't change the location of the hardware version. (Finally, if you are concerned about power consumption, you may want to change the pull-ups on the input lines to match the read values of the hardware. No need to suck even that much power.)

Types of Checksums

The simplest form of checksum is just to sum all of the bytes in question, ignoring any overflow. If your buffer is {10, 20, 40, 60, 80, 90} then your checksum should be 300. If you are only using an 8-bit checksum, that becomes 44 (300 mod 256). There are many other combinations that could sum to the same value (i.e. {44} or {11, 11, 11, 11, 0, 0}) but those aren't likely to happen by chance (statistics gives us that there is a 1 in 256 possibility if all of the bytes are random). Even this paltry 8-bit checksum is likely to save you if a byte or two gets corrupted. (As long as two bytes don't get corrupted in ways that cancel.)

If you have lots of data, 1 in 256 isn't very good odds. You could sum everything as 16-bit words which gives you a much lower probability of good checksums with bad data. Note that you need to sum the data as 16-bit words, not just keep the overflow from an 8-bit sum.

The types of errors you get depend on the memory or communication pathway. In an EEPROM, it tends to be a sticky bit that doesn't change upon command. A single byte

error will always be caught in a simple check sum. However, it cannot detect when two bytes in the stream are swapped. And errors can cancel each other out in the checksum if multiple bytes are modified. In many communication methods, the errors tend to be bursty so that many bytes will get corrupted at once.

So when people say “checksum” they often mean “CRC value”. CRC stands for cyclic redundancy check which you don't need to remember. Do recall that CRCs give you more protection than a checksum against multiple-byte and swapped byte errors. However, CRCs require more computational power than the simple sum. There are many versions available online, though I like the one [here](#) as it walks through several code examples, showing you how it works (and how to make it faster).

Remember, the checksum goal is to detect that an error has occurred. A more complex scheme might be able to tell you where the error occurred or even correct small errors.

Authentication and Encryption

Checksums and CRCs are not preventions against hacking. That is, they don't protect against intentional modification of data. There are two main forms of protection.

Encryption ensures that no one can read (or modify) your data without having the correct keys and method of decryption.

Authentication lets you know that your software is talking to something in particular. Printer manufacturers want to make sure the printer cartridges come from a known source (partially because the wrong ink could ruin a printer, partially because consumables are profitable). The authentication may be an encrypted signature so the two are often related.



Consumables are often subject to cloning but an internal database usually prevents against seeing the same one multiple times.

Authentication and encryption are difficult. I mean, the next chapter is about updating code and that is hard to understand but once you get it, you'll be able to implement something successfully. But authentication and encryption are things you can never do successfully. Sure, you might implement the algorithm correctly, but that doesn't mean the data is safe.

The more valuable your consumable or data is, the more likely someone will take the time to reverse engineer it. A determined hacker will eventually succeed, though only sometimes by a brute force attack (it is probably easier to get into the manufacturing building and access to the source code than figure out 128 bit AES key).

You can't be in charge of locking the door; the best you can hope for is that your code is robust enough to fend off the casual attacker and create enough headaches that they

leave your product alone. You probably don't have a lot of processing power for your system. Trying to implement AES or some other well understood encryption when you are running an 8-bit processing at a few MHz may make you want to help the hackers.

However, not all encryption methods have symmetric processing burdens. Some algorithms make it easier to decrypt than encrypt (or the other way). Let the host do the hard math. Alternatively, only run encryption on the critical data, leaving the rest in clear text (or with a reduced encryption level). It may make your code more complex but if it is between good encryption for the important part or mediocre encryption for all, well, I know which I'd take.

Even if you have to use an encryption algorithm that isn't good enough to stand up to the NSA (what is?), use a well understood algorithm. For all that you may have a tricky mind, your 8-bit encryption solution will not be better than a well-understood algorithm. Hiding things through obscurity instead of security doesn't protect against someone who wants your information. RSA, DES and AES are the most common algorithms. You may have to use a smaller key for an embedded system, but then you can estimate the time it takes to crack and give your management a reasonable effort level, something you may not be able to do if you design your own algorithm.

As you design your system, put on a black hat. What would you do to attack the system? Assume the hackers know the algorithm but not the keys.* Is the easiest way to obtain the keys the relatively difficult matter of putting the chip under an electron microscope to read out the code? Or can they read it from your email because another developer sent it to you in clear text? Or log into your source code repository with the guest account? Your authentication and encryption algorithms are only as good as the weakest link.

Mild paranoia aside, protecting your system is hard. Your management team will need to determine how much time (and money) they are willing to spend on developing protocols to keep your data (or consumables) safe. It is a business decision as well as a technical decision. It is also a long term decision; staying ahead of the hackers is an ongoing concern, not a one-time algorithm choice.

Changing Data

In the LCD section, I described a way to store character and image glyphs, retrieving them as needed to put on the screen. There is a design pattern that matches that description: the *flyweight pattern*. The way I described it mashes it together with a *factory pattern* so it is more correctly be called a flyweight factory pattern. Let me untangle the two so you can see them separately and be able to use the concepts outside the LCD.

* In cryptography this is called Kerckhoffs' principle: A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

The flyweight pattern's typical example is a document with many characters. The characters are objects describing their size and encapsulating a bitmap. In a non-embedded world, there is some temptation to instantiate each character object on its own. Eventually, when there are enough characters, this leads to sluggish behavior. In an embedded device, we wouldn't have that problem as there is never enough RAM to let us get into trouble this way. The solution is to point to a single instance of the character object (which for us was in flash memory).

That is it. That is the pattern: when you have lots of things used repeatedly, create a pool of instances (one for each object) and point to the pool instead of instantiating directly. Each of the shared objects is called a flyweight (like the lightest weight in boxing, not like the weight used to increase inertia in flywheels). Each flyweight must be relatively interchangeable with the others, though they may describe their state (i.e. a character may describe its width).

The factory pattern is a bit more complicated, centering on the factory method. Remember the UNIX driver model from [Chapter 2](#)? Almost all drivers implement the same interface (`open`, `close`, `read`, `write`, `ioctl`). If you are writing a program to use a driver, you open it as though it was a file ("`/dev/tty0`"). When it actually gets instantiated (returned to you), on the surface it is a generic handler but underneath, it is a serial port. The factory method (`open`) knows how to create a specific widget (subclass) that is of a generic type (class). All of the specific widgets implement the same interface (the class interface).

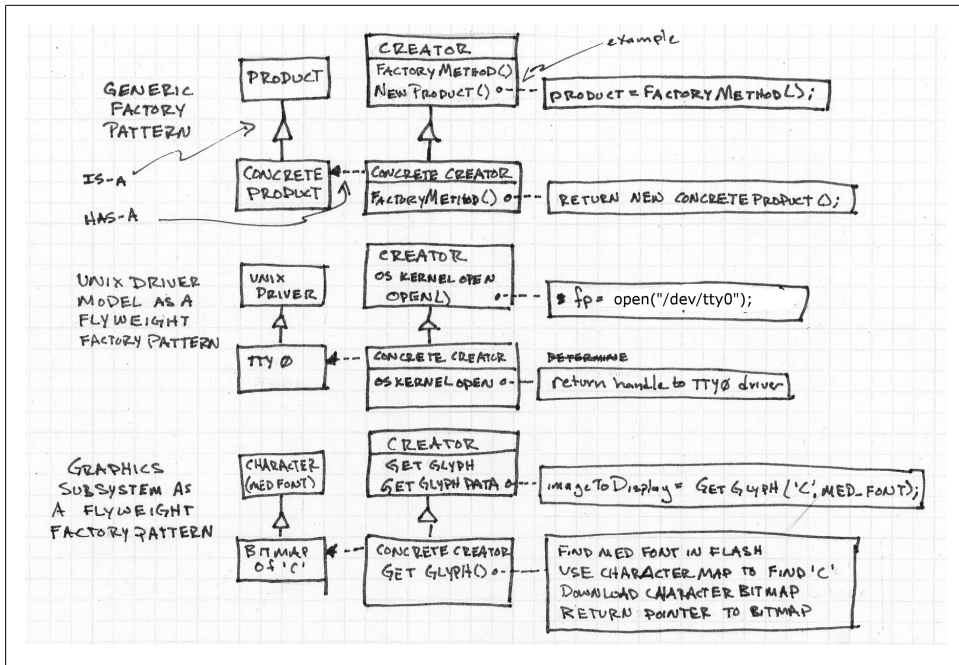


Figure 6-15. Factory pattern and flyweight factory pattern examples

The factory pattern is often implemented in a very object oriented way with specific subclasses being instantiations of a class. The creator class (the code that calls the factory method) controls the whole thing. Figure 6-15 shows the classic factory pattern diagram. Next it shows the UNIX driver model example and the LCD graphics example.



The diagram is in UML, a useful visual language for describing software, particularly object oriented software. It can be simple (like this) or far more complex, going so far as to become an interpretable language.

The factory pattern is all around you, decoupling instances from a generic ideal. It is a fairly physical pattern and the name makes sense. Imagine you have a real live factory. It is flexible enough to make any one of several, relatively similar products (the product class). In the factory is a machine (the creator). Today, you configure the machine to make sprockets (concrete product) by putting a template (factory method) into the machine. When you push the “on” button, the factory will generate sprockets. Tomorrow, you may switch out the template to make gadgets (a different concrete product) ensured by a different template (factory method).

When we do this in graphics software and with flyweights, pressing the “on” button is more like the software requesting a glyph. With a flyweight factory pattern, the factory

part manages the flyweights, making sure they are shared properly, decoupling characters from the way they are displayed on the screen.

Changing Algorithms

Sometimes it isn't the data that needs to be selected depending on the situation, sometimes the path of your code needs to change based on the environment.

If you are thinking that we saw a way to change what the code is doing based on commands from the host in the command pattern in [Chapter 3](#), you are correct. The command pattern is used to make short term command/response changes. The *strategy pattern* is used to make longer term changes, usually to data processing. According to the official definition, the strategy pattern is used to “define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.”

Let's go back to the data driven system with the ADC digitizing data to be attenuated and then sent back to analog via the DAC as shown in [Figure 6-2](#). What if you weren't sure you wanted to attenuate the signal? What if you wanted to invert it? Or amplify the signal? Or add another signal to it?

You could use a state machine but is a little clunky. Once each processing pass, the code would have to check which algorithm to use based on the state variable.

Another way to implement this is to have a pointer to a function (or object) which processes data. You'll need to define the function interface so every data processing function is the same. But if you can do that, you can change the pointer on command, thereby changing how your whole system works (or possibly, how a small part of your system modifies the data).



Some embedded systems are too constrained to be able to change the algorithm on the fly. However, you may still want to use the strategy pattern concepts to switch algorithms during development. Consider how the strategy pattern helps you separate data from code and enforces a relatively strict interface to different algorithms. Sometimes “on the fly” in highly constrained systems may include recompiling and downloading the code. Then it is more a matter of making the changes quickly in terms of releasing the code.

A related pattern is the *template pattern*. A template provides a skeleton of an algorithm but allows steps to change (without changing the algorithm's structure). Usually these aren't function pointers; the steps are part the organization of the algorithm. In our data driven system, we could make a template that looked like:

```
class Template {  
private:  
    struct sCircularBuffer *cb;
```

```

public:
    enum eErrorCode sample();
    enum eErrorCode processData();
    enum eErrorCode output();
}

```

Even though each of these functions has the same prototype, they aren't interchangeable like a strategy pattern is. Instead, they provide an outline of how the system works. An instance of this template for the system described would have the ADC sampling the data, the data being amplified and then output via the DAC. The instantiation of a template may override certain (or all) parts of the default implementation. If the ADC-process-DAC is the default implementation, a test version of the class may override the sample function to read data from a file while leaving the other two functions in place.

You can, of course, combine patterns. Here is a strategy pattern inside the template's skeleton.

```

class Template {
private:
    struct sCircularBuffer *cb;
public:
    enum eErrorCode sample();
    enum eErrorCode (*processData)();
    enum eErrorCode output();
}

```

In object oriented software, there is the concept of *inheritance* where an instance *is-a* concrete version of an object. Template patterns are like that (it is a series of these steps). Then there is the idea of *composition*, where the software *has-a* concrete version of an object. Strategy patterns are more like that (it has a function to call). Composition is more flexible than inheritance as it is easier to switch what things have, than what they are. On the other hand, building (composing) a system at run time may not be a good use of limited resources. Balance the trade offs for your system.

Further Reading

This section covered many things so I've got lots of places for you to look if you want to dig deeper into a particular area. Going backwards, as mentioned elsewhere in the book, these two references are great for understanding design patterns:

- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2. The canonical text.
- Freeman, Eric T; Elisabeth Robson, Bert Bates, Kathy Sierra (2004). Head First Design Patterns. O'Reilly Media. ISBN 0-596-00712-4.

For encryption and authentication, there are many resources. For a relatively fluffy introduction, there is a history of cryptography in WWII (almost a spy novel): Marks,

Leo (2000). *Between Silk and Cyanide: A Codemaker's War, 1941-1945*. Free Press. ISBN 0-684-86780-9.

For a more comprehensive look at security and some of the chips used to keep information secure, look at Rankl, Wolfgang; Wolfgang Effing (2010). *The Smart Card Handbook*. Wiley, 4th ed. ISBN 0-470-74367-0. This is a weighty tome but it is surprisingly easy to read, even in the security chapter.

For an example implementation of FIFO vs DMA, look for James Lynch's [write up on Atmel's AT91SAM7 serial communications](#). If you are using the AT91SAM7, look for his excellent [tutorial on setting up open source tools](#).

When coming across any new serial interface, I tend to hit Wikipedia first. There are so many variants of different protocol. The datasheet for a device may be filled with acronyms and complex timing diagrams. But Wikipedia gives me an overview that puts it in perspective.

For motor control and other control schemes, I like Dutton, Ken; Steve Thompson, Bill Barraclough (1997). *The Art of Control Engineering*. Addison-Wesley. ISBN 0-201-17545-2. It is a little more math filled than it needs to be but still pretty useful in actual implementation.

Signal processing and motor control have a lot in common in the underlying math (Laplace and Fourier transforms). If you haven't had those in school (or cannot dredge up the memories), you might consider taking a class. It isn't the sort of thing to learn from a book. But if you are going to anyway, the seminal textbook is Oppenheim, Alan V; et. al (1983 or 1996). *Signals and Systems*. Prentice-Hall. ISBN 0-138-14757-4.

Or you can skip to practical application with a really fantastic book: Smith, Steve (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub. ISBN 0-966-01763-3. It is also available [online for free](#).

Interview question: Choosing a processor

How would you go about choosing a processor for our next generation platform?

This question is multifaceted, starting out with a basic check of the interviewee's listening skills. Does he know what we build? Is he interested enough in the company to spend some time doing research? Has he been awake while I blathered on about the current system?

From there, it moves into checking their design skills and experience level. I want the candidate to know which questions to ask (What are the bottlenecks on the current platform and how have we identified them?). He should specific questions about the types of processing and the bandwidth requirements of the system to understand the problem better.

Most companies don't want to go backwards in their feature set so it is relatively safe to start with the assumption of more processing power, more RAM and more code space than the previous design. However, I want the candidate to check these assump-

tions. Ideally, as we talk about these, the candidate should describe ways to unburden the processor (peripherals often taking up an unreasonable amount of processor overhead).

A good candidate will also consider the underlying goals of a new platform (i.e. Why are we looking for a new platform?). A candidate with a product view is often a better fit than someone whose scope is more limited. To that end, I hope he asks about the volume and the cost goals. An excellent candidate will also ask if the requirements are expected to change significantly over the development cycle, helping him determine if he wants a general purpose processor or if he can choose something very specific.

Processor selection is difficult, there are many factors. I don't expect that someone can come in for a day's worth of interviews and produce a suggestion for a new processor. Most people don't keep the processor lines from the various microprocessor vendors in their head. If he immediately suggests something concrete, I take it as an indication of what he's been working on. At that point, I tend to ask why he thinks that is a good path for our particular product. The goal is to get him to tell me how he would choose, not the actual choice. In interviewing, the methods people use to confront problems is far more interesting than the solutions they generate.

Updating Code

When your system has problems after it goes out into the wild, you'll need to be able to update the code on a device in the field. To understand the complexities, let's start by looking at how uploading works in the comfort of your development lab, and how that differs from uploading to a device you've released.

During development, your system lets you peer into the guts of the code, maybe with a JTAG that can stop the processor. As part of the process, you also get to modify the code (aka *flashing the chip*). The method for doing that is chip-dependent, but it usually requires some special debugging widget. If you want to update the code after the system leaves your desk, you'll need to plan ahead. Although we'd like to arm the world with development hardware, it isn't cost effective (sadly).

The lowest level of updating code is very chip-dependent, but follows certain common patterns.



There are many different names for updating code on a device: bootstrap loading, bootloading (even when it isn't related to booting), updating, uploading, burning the code, and over-the-air programming. Whatever you call it, uploading code is one of the most difficult topics in embedded systems. It is right up there with storing code on cartridges, keeping consumables secure, and faking floating point math via binary scaling. This chapter may be tough the first time you read it. Don't lose hope.

Three different loaders are described in this chapter in increasing order of complexity. Some issues you should know about before we start include:

The code storage mechanism and communication method

We'll need something to hold the new code image. What you employ to store it on depends mostly on the communication method, for example: a thumbdrive when updating over USB, an EEPROM for SPI, or a hard drive for a network.

Code space memory (old code location)

This is often called ROM (read-only memory). However, since we are about to change its contents, it isn't read-only. Instead, the code space is some sort of non-volatile memory (memory that is not erased when the power is lost) that often requires special voodoo to rewrite. Once this really was read-only memory and firmware updates simply weren't possible at all. Now this type of memory is usually flash and able to be rewritten so that runtime code can be updated in the field (as well as on your desk or in manufacturing). Whether it is flash or some other form of ROM, writing to it is more complicated than writing bits to RAM so it is often called *programming* the memory. How you program the memory where your code is stored depends on your processor.

Scratch space RAM

Ideally, this RAM is the same size (or larger) than your runtime image. It should be local to the system. While not strictly necessary, the scratch space prevents serious complications if communication with the new code is lost before programming is complete. The idea is to write the code to the scratch space and make sure the upload is complete without corruption before actually copying it over to where it counts—the permanent memory of the device.

Run space RAM

This is RAM from which the processor can execute. If you don't have this, it limits your loading options.

On-Board Bootloader

Some processors have an internal bootloader that will load code from an external source if the right I/O pins are set. In an ideal world, you'd just set the I/O pin while you connected the new code to the system, generally as the system powers up. The bootloader would automatically load the data into the code space.

Part a) of [Figure 7-1](#) shows the new code connected to the processor using one of the communication methods mentioned earlier. The bootloader code inside the chip reads in the data and writes it to the code space.

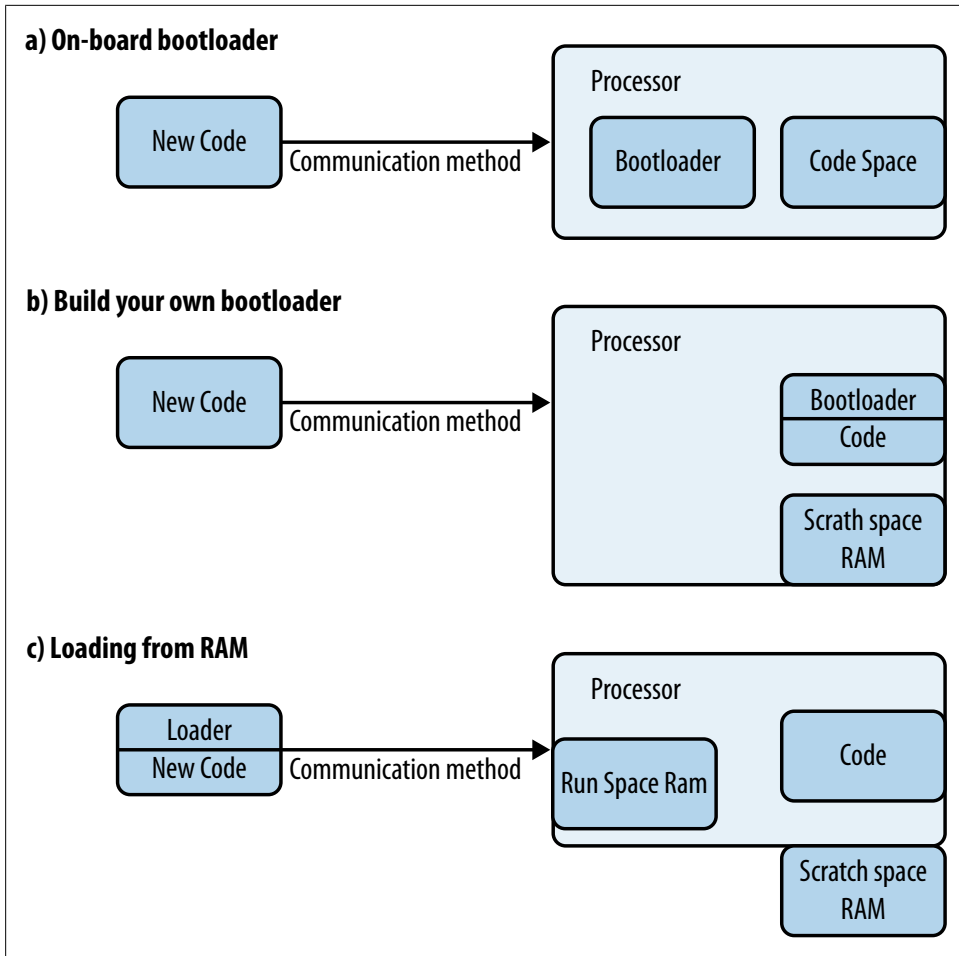


Figure 7-1. Three architectures for loading code

If that figure describes your system—excellent. If you can make that be your system, do so. Everything else is a lot more complicated when it comes to uploading in the field.

Build Your Own Bootloader

If you have plenty of code space, the next easiest option is to build your own bootloader: a resident program in the code space that could reprogram the rest of the memory. Since most code space is flash memory, you will have to erase sectors of the old runtime code before you can write the new code.

The naive bootloader would erase a sector, read the new code into some scratch memory and use chip-specific functions to write the code to executable memory, repeating until all runtime sectors are complete. What could possibly go wrong?

First, the new code could come corrupted from the source. Second, the communication method could drop part of the message. Third, the memory holding the new code could be removed. A checksum could detect these problems, but that won't help if the bootloader has already written the data to the code space before it can detect the error. Programming a bad image will lead to a non-functional unit at best. The scratch space RAM is optional but it will help you avoid those pitfalls. See part b) of [Figure 7-1](#) for a high level view of this system.

Your bootloader is a resident of your code space. This gives you the opportunity to recover from failures. Even if there isn't enough RAM to validate the whole image, so that you have to take the risk that a corrupted image will be installed on the device, it will be OK as long as you allow part of the bootloader to run during the power-on sequence. The bootloader can verify that the runtime code image is valid, by running the checksum that was uploaded with it. If the code isn't valid, the bootloader can wait as long as it takes for you to connect new code. Yes, the device is non-functional, but that's probably better than having it display garbage or short out some part of a factory system. This solution does mean that your bootloader has to be the first code to run at boot time, as shown in [Figure 7-2](#).

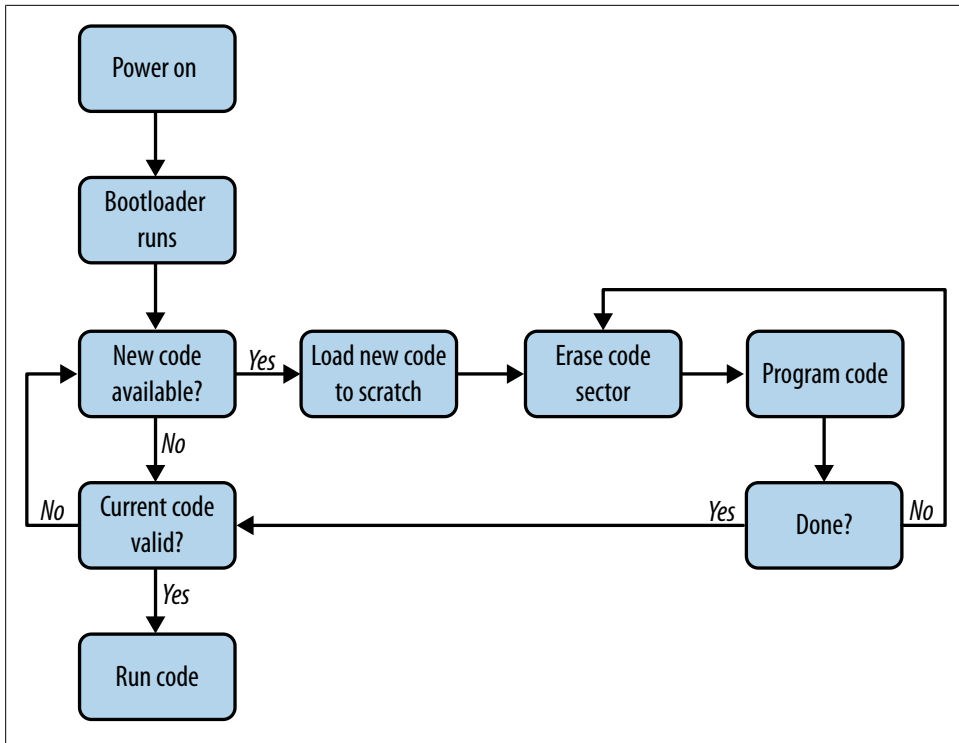


Figure 7-2. Build your own bootloader flowchart

Since the bootloader will be erasing the rest of the code, it must contain everything it needs to run. Instead of just a function to be called, it is a mini-program embodying code to communicate with the new code storage mechanism, processor-specific functions to write to the code space, and any debugging code you need to make it all work.

If the code space can only be erased in sectors, the bootloader must take up a whole sector (or a whole number of sectors). If you are a little short on code space, you might consider putting other things in the section with the bootloader (for instance, fixed lookup tables). These sectors become untouchable, never to be updated, so you have to hope any problems that arise don't reside there. Test your loader thoroughly.

Modifying the Resident Bootloader

If you really need to, you *can* modify the bootloader using a two-pass process that resembles a shell game. See the table to identify where the bootloaders and runtime codes go. The asterisk (*) indicates which code is running at each stage.

Stage	Storage mechanism	Bootloader area of memory	Runtime code area of memory
1	BL2	Old bootloader*	Old runtime

Stage	Storage mechanism	Bootloader area of memory	Runtime code area of memory
2	(Nothing)	Old bootloader	BL2*
3	New bootloader	(Erased)	BL2*
4	(Nothing)	New bootloader	BL2*
5	New runtime	New bootloader*	(Erased)
6	(Nothing)	New bootloader	New runtime*

In your new code storage mechanism, put an image that contains code similar to the bootloader (we'll call it BL2). The old resident bootloader will load this and then run it. BL2 erases the old resident bootloader and writes a new resident bootloader from the storage mechanism. Once the new resident bootloader runs (this may require a reset), it can load a new runtime image. See the flow of control in [Figure 7-3](#), note each stage from the table is marked.

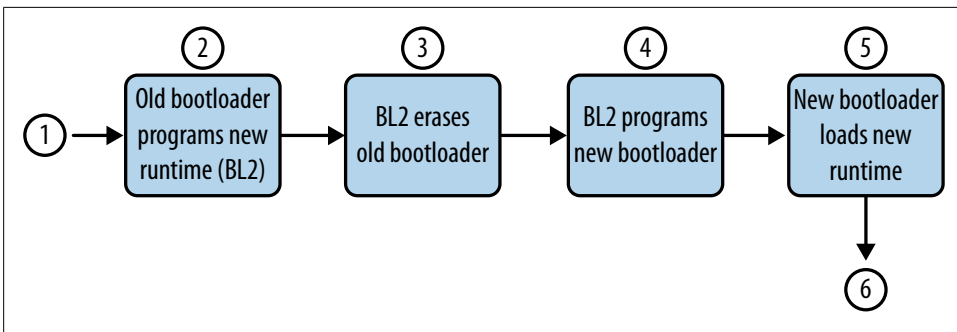


Figure 7-3. Modifying the resident bootloader

If you can leave the bootloader resident in the code space, building your own can be a good solution, workable for many systems. However, this method doesn't work if you don't have enough code space to devote to the bootloader, if the bootloader functionality changes regularly, or if the code space doesn't allow sector erases (a security feature).

Brick Loader

The next loader solves all of the problems mentioned at the end of the previous section but is more complicated and riskier. The danger comes from the possibility of making the system useless and unable to ever load valid code (aka "turning the system into a brick").

There is a period between erasing the flash and having the new code fully loaded on the system. If the power is lost during this time, the system is not recoverable in the

field. Depending on the processor, the system may never be recoverable. One of the goals of good loader design is to make this period as short as possible.

Let's get back to our processor and its new code. We've determined that the loader can't run from the normal code space because we are going to erase that and put in new code. That leaves the processor RAM as the only place left to run the code from (though some architectures disallow this so not all processors can have this type of loader). See part c) of [Figure 7-1](#).



Loading code isn't the only time you may want to run from RAM. Sometimes it is faster! See [“Memory Timing” on page 240](#) for more information.

In [“Build Your Own Bootloader” on page 201](#), the bootloader was separate from the rest of the program because it had to contain everything it needed to operate while the runtime code was erased. For the loader in this section, you'll need a completely separate image of the loader code. You will also need to edit the linker script to make the loader run from RAM. Linker scripts are not for the faint of heart, but they are reasonably standard across platforms. Once you learn the basics of one linker script language, others will be easier to use. And you don't need to learn all the features, just enough to modify an address and segment length here and there.

To get the new runtime image running using our loader, we'll need to implement five steps:

1. With the runtime code, copy the loader from the new code storage to RAM.
2. Run the loader code.
3. Copy the new code to a scratch area.
4. Erase the old code and program the new code.
5. Reset the processor to run the new code.

In this section, I've dropped the “boot” from bootloader. Where the methods in the previous sections are part of the system and can be part of boot time checking, the loader in this section only runs for a short time.

Linker Scripts

After your source code gets compiled and assembled, the object files (and libraries) are combined by the linker into an executable file. The resulting code has three sections:

bss segment

Contains uninitialized global variables. This will go in RAM. The odd name has historical reasons that don't concern us here.

Data segment

Contains global variables that are initialized. This will go in RAM. The data segment may include bss as a subsegment. It may also include the heap and stack.

Text segment

Contains code and constant data. This may be put in read only memory or in RAM.

Vector segment

A special part of the text segment that contains the exception vector table to handle interrupts.

The linker reads a text script to determine the memory layout of the output. To move the code or place buffers at certain addresses, you'll need to modify the linker script. Don't write one from scratch. When you build your executable, it already has a linker script, usually ending in *.ld*. Find the existing one and modify that. (Or look for a loader example specifically for your processor and start with that one.)

For example, if our system has flash for storing and running code, some internal RAM, and some external RAM, the memory map might look like:

Address	Size	Memory type	Segments that can be placed here
0x000000	0x0FFFF	Read only memory (flash)	text
0x010000	0x07FFF	Internal processor RAM	text, data and bss
0x110000	0x1FFFF	Off chip RAM	data and bss

A simple linker script representing that would look like:

```
SECTIONS
{
    /* Memory location is in Flash, place next commands at this location */
    . = 0x000000;
    Code : {
        *(.vectors)} /* Put interrupt table at very first in memory */
        *(.text)
    }
    /* Now put everything in off board RAM */
    . = 0x110000;
    Data : { *(.data) }
    UninitGlobals : { *(.bss) }
}
```

The script is very order-dependent. The line `. = 0x000000` indicates that the cursor (where the next section will be placed) is at 0x000000. The next line `(.text : { *(.text) })` says to put that part of the text segment in a section called `Code`.

A more complex linker file would define the size and addresses of the memory so that the linker could give an error if the segments didn't fit in their allotted spaces. Also note that the type of memory (readable, writable, or executable) is specified to enable more error-checking.

```

MEMORY
{
    /* Define each memory region */
    Flash (rx) : ORIGIN = 0x000000, LENGTH = 0x0FFFF /* 64k */
    InRam (rwx) : ORIGIN = 0x010000, LENGTH = 0x07FFF /* 36k */
    ExRam (rw) : ORIGIN = 0x110000, LENGTH = 0x1FFFF /* 128k */
}
SECTIONS
{
    Code : {
        *(.vectors)}
        *(.text)
    } > Flash
    Data : { *(.data) } > OutRam
    UninitGlobals : { *(.bss) } > OutRam
}

```

Linker scripts can get very complicated. Don't get bogged down in the details; focus on the addresses and how they match the table. Look up the language, start out by making small changes, and look in the program's output map file to see the effects. (Reading a map file is discussed in [Chapter 8](#).)

As with many scripting languages, it is pretty easy to build up a linker script that is so complicated no one can decipher it. Be careful. If you have a gnarly linker script to start with, search the Internet for “linker script” for tutorials and manuals for more detailed information.

When creating a RAM-based program such as a loader, you'll need to move the code from the Flash section to the InRam section (or in the first example, set the initial memory location to the internal RAM address, `.= 0x010000` instead of `.=0x000000`).

Creating buffers at hard-coded addresses is only a little bit trickier. You can add variables before any sections (before `SECTIONS` or `MEMORY`). Then use those variables in the script (and in your code).

```

_linkSensorDataLen = 0x1FFF;
_linkSpiBufferLen = 0x2FF;
_linkMaxLoaderSize = linkSensorDataLen + linkSpiBufferLen;

MEMORY
{
    ... /* same as before */
}
SECTIONS
{
    Code : {
        *(.vectors)}
        *(.text)
    } > Flash
    /* The special buffers allocated here, the NOLOAD indicatess to the linker that
    ** it doesn't need to do anything with these gaps in the memory */
    Special (NOLOAD): {
        _linkSensorData = .; /* address of data buffer */
        . = . + linkSensorDataLen; /* increment memory pointer to leave a gap */
    }
}

```

```

    _linkSpiBuffer = .;
    . = . + linkSpiBufferLen;
} > InRam

Data : { *(.data) } > OutRam
UninitGlobals : { *(.bss) } > OutRam
}

```

As for using those linker variables in your code, it depends very much on your compiler. Here is one way I've seen it handled. The `_linkSensorData` variable comes from the linker script, and the program just creates an ad hoc struct called `sSensorData` in order to get access to the data at that point:

```

extern unsigned long _linkSensorData;
extern unsigned long _linkSensorDataLen;
struct sSensorData *gSensor = (struct sSensorData *) &_linkSensorData;

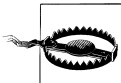
```

Copy Loader to RAM

The process just described sounds pretty simple, but a few details make it difficult. First, you can't just copy the loader into RAM that your runtime program is using. If you put the loader in the same RAM used by the runtime (including interrupts), your program will exhibit random behavior and crash. You'll need to allocate some RAM specifically for the loader code—enough for the whole loader program—and make sure the runtime program doesn't use that RAM for the stack or heap.

Next you have the problem that your loader program is built to run at a certain address (its base address), calling functions at certain other addresses. The RAM you allocate has to be at that particular base address, which is not something that you can do in standard C or C++. Compilers for embedded systems will give you a way to do this, sometimes using the `@` symbol, using a `#pragma`, or through the use of linker variables. If the solution involves linker variables, you will need to modify your runtime linker script to set the address of the RAM buffer and create a gap the size of your loader program.

This is pretty compiler-specific and processor-specific stuff. “[Linker Scripts](#)” on page 205 shows some sample linker code. Yours won't look exactly like that, but it will give you a place to start.



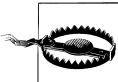
Loading the RAM-based executable at runtime takes a bit of arrangement. Before getting all gung-ho about writing the loader, start small and create a program that blinks an LED (or something similarly hello world-esque). Get it running from RAM, ideally via a debugger (though some debuggers do not support this well, one of the reasons I'm suggesting you start small, in easily debugged chunks). Then get your runtime code to copy the blinky test program from the new code storage mechanism to RAM and run it.

If your loader program is greater than the size of your available RAM, start by divesting the loader program of all unnecessary functionality. If it still doesn't fit, programming new code in the field may require an on-processor bootloader. But don't lose hope yet. Is there a large buffer or set of medium-sized buffers in your system? (For instance, you may be able to turn up some display or data acquisition buffers.) If the code is about to have its mind wiped, these buffers may not need to stay valid until the brain transplant is complete. In fact, if your program stops executing all interesting functions, you might be able to free up enough space for the loader to execute from. Make a list of the buffers that aren't needed during the firmware update. If the loader code can fit in the sum of those buffers, you can make this work.

In the linker script, put your large buffer at the top of RAM (the lowest address). If the loader program area contains multiple buffers, you'll need to allocate them and specify their sizes. At the end, you'll also need to figure out exactly how big your loader can be.

For example, if we use the system described in the linker script, we have our flash memory at 0x00000 where the code runs from. We have RAM at 0x010000 which is where we'll want the loader to go. Let's say our system has some sensor data and a circular buffer for communicating with the sensor. At the end of updating the code, the system will reset and lose the sensor data anyway. So during the loading process, we can halt the subsystem that uses those buffers and reuse them. The loader will run from the space that once held the sensor and SPI data. The entire loader program must fit in that space.

Address	Buffer	Buffer Size	Loader Ramifications
0x010000	Sensor 1 data	0x1FFF	Loader base address should be 0x010000 in its linker file.
0x012000	SPI circular buffer	0x2FF	
0x012300	(No more big buffers)		Loader program can be no more than 0x2300 in length.



Don't forget to disable all interrupts that use any of this memory before loading your bootloader there.

Run the Loader

Once the loader is in RAM, you'll need to run it. Essentially, you want to blindly execute the code loaded in RAM.

This is kind of scary. One way to make it less scary is to be sure that the loader is correct and complete. As you build the loader, calculate a checksum. Then calculate the loader checksum as you load the code into RAM. If they don't match, log an error and reset the system. The old code will still be around and a reset will clean up the RAM so that the runtime code can safely use it.

Once you get down to actually loading the code, you'll need to use a function pointer ([“Function pointers aren't so scary” on page 66](#)). The function pointer should be set to the address in RAM that holds the loader program. Once the function pointer executes, it will never return.

```
int loaderStartAddress = 0x10000; // should come from the linker file
typedef void (*tFunctionPointer)(void);
tFunctionPointer fp = (tFunctionPointer) (*(uint32_t*)(loaderStartAddress));
(*fp)(); // jump to loader, never to return
```

If you are walking through the code, this jump will make your debugger lose all context so that it will show you only assembly code. Some debuggers allow you to reload symbols without reloading code, which will fix the context. In others, you'll have to test the process out first with a small, easily debugged test loader (i.e. blinking LED).

Copy New Code to Scratch

Once the loader code is running, the next step is to get the new runtime code on the system. You could erase and program the new code directly, but if something happens, you could end up with a system whose only program is a RAM-based loader. Once the power is cycled, you end up with a brick.

Unlike the procedure for building your own bootloader, here you don't have a back-up way to check and load code at power-on. These failures are much worse than having a unit that doesn't *currently* work. They lead to units that can never work again. The optional scratch RAM becomes more critical for this type of loader due to the more dire consequences.

Once the image is good in its entirety and the image cannot be removed due to user impatience or network recalcitrance, the old code can be erased and the new code programmed.

Dangerous Time: Erase and Program

Up to this point, no damage has been done to the system. If the system resets, the worst that can happen is a break in normal operation. However, once you erase the old code, that's no longer true. Nearly anything that goes wrong is a catastrophic system failure, including a power loss.

So before erasing the old code, disable all interrupts (really this time, no excuses). Then write the new code. A reset will make the new code run (a good excuse to stop feeding the watchdog (see [“Watchdog” on page 143](#) for more information about using your watchdog timer to improve system robustness)).



Chip vendors often supply the erase and write functions, so look around before writing them.

Reset to New Code

The flow chart in [Figure 7-4](#) shows the process we've gone through to get here, with the dangerous area highlighted.

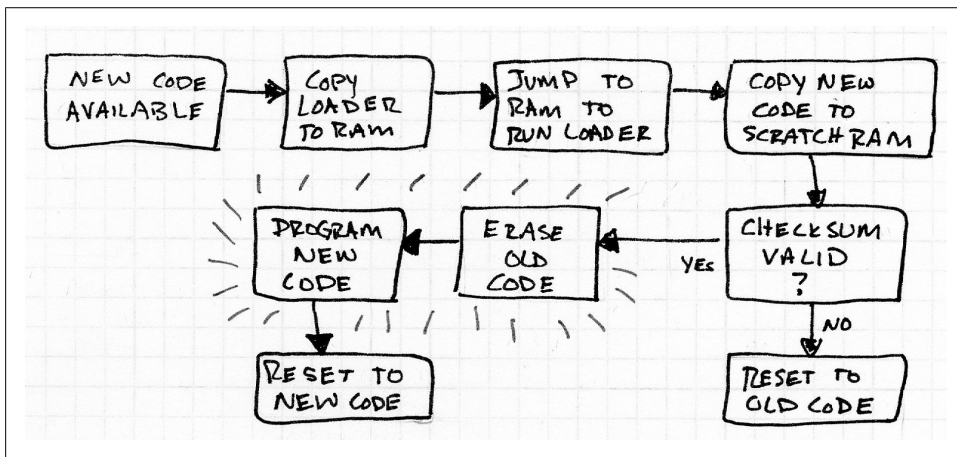


Figure 7-4. Loader flowchart

There are a few additional points of interest when thinking about loaders. First, where does the loader come from? The loader could be stored in the current image already on the system. This implies that the old code needs to know how to load new code. However, it isn't always possible to get users to update for every revision. You'd have to architect your bootloader in a way that allows users to leapfrog versions. Or you can get your loader from the same place as the new application code, as shown in part c of [Figure 7-1](#). Then, as described there, your runtime needs to know only how to copy the loader to a spot in RAM and jump to it. The loader retains control of the rest of the process, adding flexibility. As long as the loader fits in the memory allocated by any of the released runtimes, users can upgrade to any revision from any other revision.

Second, this process assumes that getting the new image is straightforward. However, if it is not so simple, such as when you need to send the image over a network, one way to keep your loader code small enough to fit in your RAM is to let the runtime handle the heavy lifting and put the loader's image someplace local before handing control over to the loader. This decreases the flexibility for future, unforeseen needs, but you may need to do it.

Finally, loading code is dangerous. Often the loader is the last piece of code written and so one of the least tested parts of the system. As you draw a flow chart for what goes on in your loader, look at the amount of time when the old image is erased and the new image is running. Minimizing that time will save your units.

Security

Adding security into the mix tends to make bootloaders exponentially more complex. In general, winning against a sea of hackers is impossible, but you can make casual attacks more difficult. The first step is to identify what you are protecting: a secret algorithm in the code? the ability to create or verify consumables? the integrity of the hardware? What you choose to implement depends on your priorities.

You might get a little bit of help from your processor. Many chips offer code read protection. Once it is turned on, on-chip memory becomes unreadable. The processor can still execute the code, but a debugger (or a loader) cannot read the code space.

Often times, the code read protection will limit the processor's ability to erase sector by sector. Instead, you have to erase all of it before updating code. (Some levels of protection won't even let you write new code at all.)

Once you've done that, the weak point is in the way new code is handled. Ideally, the new code is seen only by trusted associates who will be updating the code (or the code will travel only over secure networks). However, if you allow the user to update the system, you need to augment your loader with some security features.

If you build your own (resident) bootloader, you should encrypt the new application code. The bootloader will need a decryption method (which will increase the number of sectors to be devoted to the bootloader).

Because the loader must fit into a limited amount of space, it may not have a lot of space to authenticate your new code. If you don't need to encrypt the whole code (perhaps just secret keys for consumables), consider encrypting the loader with the keys. The runtime can decrypt the loader as it puts it in RAM. Then the loader can program the clear text new code, adding the secret keys only the loader knows.

What you do depends on your system and the goals you have. It will require some twisty thinking to determine an authentication scheme that works for you. Remember that loading new code is likely to be a weak point in the overall security system.

Summary

These three strategies are good places to start your design. There is a lot of room between the ones presented and your system requirements. For example, you may want to consider having a resident bootloader but copying it to RAM when code is updated to allow it to be reprogrammed. Or your scratch RAM may be only half the size it needs

to be so that you update the firmware in two chunks. There are lots of options, now that you understand the basics.

Questions to ask yourself about your design:

- How often will new code be loaded and by whom? Are they trusted associates? Experienced technicians?
- Would adding a low-cost part make loading new code safer, and can you include it?
- What other pieces of the system might change? Will you need to accommodate updating information on different chips? (Maybe reprogramming an external flash with new data?)
- What if the new code is corrupted? What if the new code becomes unavailable because its media is removed or communication to it is lost?
- Are there security concerns? What are the points of attack? (Authentication keys? Algorithm?) What are the vulnerabilities? (Reading code from the processor? Reading code from the new code storage mechanism? Hacking the loader and taking over the hardware?)
- At each stage, what is the worst that can happen? For the unrecoverable stages, how can you make them take less time or decrease the probability of bad things happening?

Once you determine what is important about your loader and set some goals, you can design the methodology to update the code.

Interview question: Getting a goat safely across a river

A man is taking home a goat, his partially tamed wolf, and a cabbage. They reach a river but the bridge is washed out. The boat conveniently tied on his side of the shore is very small and can hold only the man and one passenger at a time. If the goat and the wolf are left alone, the wolf will eat the goat. If the goat and the cabbage are left alone, the goat will eat the cabbage. The wolf, however, will not eat the cabbage. How can they cross the river safely? (See [Figure 7-5](#).)

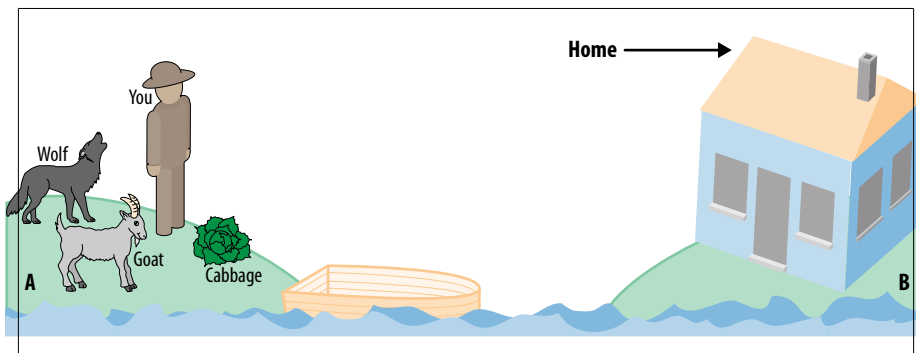


Figure 7-5. Taking everyone home safely

While this style of question is pretty common, I am strongly opposed to interview questions that involve goats crossing water so I wouldn't ask this one. Basically, if it is necessary for me to herd goat(s), I don't particularly want the job. However, loading code is a lot like this problem. There are dangers along with resource limitations at multiple points. You have to know what to look for and then use some little mental twist you learned along the way to make the leap to the solution. I generally prefer to see how people think instead of whether they are in the club that already knows the answer, so I wouldn't ask this question. Since I don't know what I'd look for; I'll just provide the solution.

In this case, the trick is to note that the goat is the dangerous one and must be kept away from the others. If you frame the problem that way, the solution falls out a little easier.

Start on side A with the wolf, goat, and cabbage. Cross to side B with the goat. Return to side A. Take the wolf (or the cabbage, doesn't matter) to side B. Now, take the goat back to side A, leave it there while you ferry the cabbage (or wolf) back to side B. Cross back to A, get the goat, cross back to B and move the parade along.

The trick behind the problem is that you can take extra trips to get the job done, as long as the job is done properly. Updating firmware is a lot like that.

Doing More with Less

Engineering requires technical skills and a deep understanding of the relevant technology. Writing good embedded software goes a step further, also requiring a devious mind with an affinity for puzzles.

Implementing the requirements on a system that has everything you need is a matter of turning the metaphorical crank to get the answer. Some solutions are more elegant than others but most will work well enough to get the product shipped. It all gets a lot more interesting when you have a system that seems as if it can't possibly contain everything you need. You can compromise on the features but where is the fun in that?

For me, the great part of embedded systems implementation is the thrill of finding just a tweak that liberates a few more processor cycles, being excited about freeing eight bytes of RAM, persevering through the map file to find a whole section that you can reclaim for the use of your code, and realizing you can get your product the coveted green award if you can just squeak out a few more milliseconds of deep sleep.

The downside to all of this is that you may make the system more fragile. For instance, when you free the RAM by having two otherwise decoupled subsystems share it, those subsystems become linked. Maintainers of the system become confused and frustrated by hidden linkages. The code is no longer modular and subsystems cannot be reused. Flexibility is lost.

This chapter looks at the ways to get more out of your system. We'll need to start by characterizing the resources we have and those we need. Some of the resource optimization techniques allow you to trade one resource for another. Identifying the plentiful resources is almost as important as determining where the scarce ones disappear to.

One of the most important resources is development effort. Your time is valuable. You will need to balance value versus the time it takes to implement a solution: moving to a larger, faster chip in the family is a cost to all units which could be a win if you are building only a few; trading memory and processor cycles may require you to restructure some code; and going line by line to optimize assembly code is going to take serious effort (and skill).

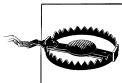
Code Space

Implementing an application on a system without enough code space is like trying to write a term paper in a booklet that is too small. Even though you plan ahead and try to figure out how much space should be allocated to each point, in the end, you will have to write tiny on that last page and wind up using the margins. This section will help you cope with the situation.

Reading a Map File (Part 1)

I get a sinking feeling when the linker gives an error message during a build. The compiler provides all sorts of friendly advice and critiques my typing. But the linker doesn't usually talk back unless it is something important.

On the other hand, the linker provides a wealth of information if you know where to look. [“Linker Scripts” on page 205](#) describes the linker input. The output map file is easier to read, though still foreign to most developers.



You may need to configure your linker to output the map file. Look in the directory where your executable is located for a *.map* file. If it isn't there, check the manual.

Map files are processor specific. The examples in this section come from a GNU based tool chain for the NXP LPC13xx. Most map files have the same information though it may be in a different order or have different formatting. You'll need to make some educated guesses as you look through yours. If you aren't sure that your map file is giving you the information you are looking for, try this: make a copy, change the code, and then diff the resulting map file with the original.

The GNU linker for the LPC 13xx starts off with a list of the library modules that are included and which of your modules is responsible for the inclusion.

```
Archive member included because of file (symbol)
../lib/gcc/arm-none-eabi/4.3.3/../../../../arm-none-eabi/lib/thumb2/libcr_c.a(memcpy.o)
./src/aes256.o (memcpy)
```

If you find that a library is large, this section helps you figure out which part of your code is calling functions in that library. You can then decide whether the code module needs to make that call or if there is a way around it.

Next in the map file is a list of global variables and their size.

```
Allocating common symbols
Common symbol      size          file
gNewFirmwareVersion
                   0x6          ./src/firmwareVersion.o
```

Limiting the scope of the variable using the **static** keyword will cause the variables to be later in the file, so if a common symbol like the one just shown isn't in your file,

you've done a good job of getting rid of global variables. If you have a lot of data here, review [“Object Oriented Programming in C” on page 26](#) to remember the difference between global variables and file variables.

Next is a list of sections, addresses and sizes of code that is not referenced by anything. This section, titled "Discarded input sections" in the LPC 13xx map file, shows which functions and variables are cluttering your code but not taking up any code space. Often the unused code remains in the code base if it is vendor code you don't want to modify or test code which is only used in special circumstances.

A reflection of the linker script is shown in a memory map:

Memory Configuration			
Name	Origin	Length	Attributes
Flash	0x00000000	0x00008000	xr
RAM	0x10000000	0x00002000	xrw
default	0x00000000	0xffffffff	

Some map files also give you the amount of each resource used. This is very helpful for determining how close you are to running out of resources.

The next section of the map is a tedious list of all the files included in your project. Skip over that.

Eventually, you get to a section that lists each function, the address it is allocated to, the function size, and the file it came from. These are in the order they occur in the compiled image, generally grouped together by file (module).

This starts with the first section in the linker file (you may want to pull up the linker input *.ld* file as you go through the map file so you know what to expect). The line at the top gives the section (*.text*, which contains all the code), the address, and the size of the section.

```
.text          0x00000000    0x7ccd
```

Other than experience and expectation, there is nothing to indicate that this line is more important than the others. In the "Memory Configuration" snippet shown earlier, the total flash size is 0x8000. Here we see that the code is taking up nearly all of that.

Next, the listing breaks down the contents of the code section:

```
.text.Initialize
      0x0000037c    0x7c ./src/main.o
      0x0000037c          Initialize
.text.main      0x000003f8    0xb4 ./src/main.o
      0x000003f8          main
```

In the map output, most functions have two lines like this, giving slightly redundant information:

```
.section.functionName  address  size      file
                        address      functionName
```

This part of the map file shows you the size of every function. The first step to reducing the footprint is to find out which pieces are particularly large. Many of the largest pieces will be libraries, especially if you are using floating-point, C's standard I/O (`scanf/printf`), or C++'s `iostream`. Even operators such as division end up calling functions.

```
.text.__aeabi_ldiv0
      0x00006c6c      0x4 ../lib/gcc/arm-none-eabi/4.3.3/thumb2/libcr_eabihelpers.a(rtlib.o)
      0x00006c6c      __aeabi_ldiv0
.text.__bhs_ldivmod
      0x00006ec0      0x20c ../lib/gcc/arm-none-eabi/4.3.3/thumb2/libcr_eabihelpers.a(rtlib.o)
      0x00006ec0      __bhs_ldivmod
```

The `aeabi_ldiv0` function is a wrapper because it is only four bytes, just enough to jump to another function. However, `bhs_ldivmod` is a real function. Doing signed long divides is costing the program 0x20c bytes, more than twice the size of the main function.



Some map files don't break out the function sizes in the same way. Instead they list the address of each function in the image and you need to calculate the differences to get the size.

Functions are not the only things that take up code space. Later in the example map file is the section for read-only data:

```
*(.rodata*)
.rodata.str1.1
      0x000070cc      0x36 ./src/main.o
.rodata      0x0000715c      0x10 ./src/aes256.o
.rodata.str1.1
      0x0000716c      0x35 ./src/aes256.o
```

Here, `main` has 0x36 bytes of `str1.1`, not something I'd name a variable. Actually, this is the automatic name this compiler gives to string variables. So wherever there are constant strings (such as “Hello world”), the compiler collects them into one area that goes into the read-only data section. In addition to the 0x35 bytes of string data in `aes256.o`, the previous snippet also shows unspecified data in the file (0x10 in length). In the source file, in the test function, a variable is declared with static initializers:

```
uint8_t buf[16] = {0xe5, 0xaa, 0x6d, 0xcb, 0x29, 0xb2, 0x71,
                  0xae, 0x0e, 0xbc, 0xfa, 0x7a, 0xb2, 0x2b, 0x57, 0x59};
```

Because this is inside a function, it isn't visible by name to other files. The map file therefore doesn't show the variable name, but lists the constants because they take up space in the executable image.

All constants take up space whether they are declared in `#define` declarations or with the keyword `const`. Global constants are often visible in the map file, but variables declared with `#define` end up in the code, so they aren't called out in a similar way. You can see the variables declared with `const` in the map file under `.rodata`.

```

.rodata      0x00007ab8      0x6A ./src/displayMap.o
            0x00007b8a      kBackgroundInfo
            0x00007bf4      kBorderInfo

```

There may be some sections that say `fill`. These indicate that the data between files got misaligned, generally because a file used a constant that was smaller than the native format of the processor. For example, a character string that is five characters long will consume 1.25 words on a 32-bit processor, causing the linker to fill three bytes (usually with zeros). This is wasted space but difficult to recover. Unless you really need every byte, it is better to focus on the larger users of memory.

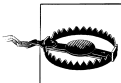
The rest of the map file is filled with juicy RAM details (we'll get to those later) and debug information (not generally useful unless you are building a JTAG unit).



If the map file doesn't present the data in a way that is useful to you, write a script to parse it yourself! Read the map file and create an output table where the height of the cell is proportional to the size of the function. This is pretty easy to do in Python, especially if you use HTML as the output.

Process of Elimination

Now that you know what is taking up your code space, you can start to reduce the size. Start with your tools. In addition to the different optimization levels you normally see to make your code go faster (i.e. the `-O3` compiler option), there are special optimization flags to make your code smaller (for instance, in GCC, `-Os` tries to optimize for code size instead of speed of execution). This may be enough to solve your code size issues.



If you get a different runtime outcome when you turn optimizations on, check that all of your variables are initialized and that volatile variables are marked as such.

If compiler optimizations aren't enough, as you look for memory, it is useful to keep score so you know which changes lead to the best improvements. Not only does this give you a feel for the types of improvements, it will help you express trade-offs to your colleagues (e.g. "Yes, that code is uglier now, but it saves 2Kb in code space.").

As in [Table 8-1](#), create a spreadsheet starting with the baseline numbers. For every change, fill in a row so you can see the relative value of the changes. If your linker output gives you the numbers in hex, you may want to let the spreadsheet translate to decimal (or the other way around).

Table 8-1. Optimization Scorecard

Action	Text (code)	Data	Total	Total (hex)	Freed	Total freed
Baseline	31949	324	32273	7E11		

Action	Text (code)	Data	Total	Total (hex)	Freed	Total freed
Commented out test code	26629	324	26953	6949	5320	(Reverted change)
Re-implemented abs()	29845	324	30169	75D9	2104	2104
Calculated const table at init time	29885	244	30129	75B1	40	2144
= comment from you	= size of .text section	= size of .data section	= total image size	= hex of total image size	= bytes freed with this change	= total bytes freed since start

The table documents how you tried multiple actions, including commenting out a large block of test code. As described in [Chapter 3](#), it is often good to be able to run code tests in the field. However, these take a huge chunk of precious space. While feature reduction is undesirable, when looking at code space reduction, the relative importance of features should be kept in mind.



“Reduced size of code by 40%” is a super line on your resume. Hard numbers are great but be prepared to explain how you did it and the trade offs you considered. (And the explanation shouldn't consist solely of “turned on compiler optimizations”!)

Libraries

As you go through the memory map, look at the largest consumers first. You may find some libraries are included that you don't expect. Trace through the functions to see where the calls to these libraries are coming from.

While some (monolithic) libraries are included if any function is used, other libraries are granular, loading only the functions required. Even the standard libraries can be monolithic so that using the built-in string copy function leads to a large footprint. Your map file will show you these space hogs.

Many times you can write a function to replace the library. Other times, you'll need to figure out how to work around a limitation. Here are some common examples to get the ideas flowing:

- Replace floating point numbers with fixed point representations (see [Chapter 9](#)).
- Replace `printf` with a few functions that don't take variable arguments (`Log`, `Log WithNum`).
- Replace `strcpy` with your own implementation to exclude the strings library.
- Replace the `abs` function with a macro to remove floating-point math library dependencies.

Functions and Macros

Keeping your code modular is critical to readability. However, each function comes with a price, increasing code space, RAM and processor time. The code space cost is easy to quantify.

For example, I needed to find a small implementation of a way to find the minimum of three variables ([“Functions and Macros” on page 221](#)).

Example 8-1. Minimum of Three Algorithm

```
if a < b,  
    if a < c, return a  
    else, return c  
else,  
    if b < c, return b  
    else, return c
```

I put together some code to try out the different implementation options. First, I wrote out the code in the main function and compiled it to get my baseline code size in bytes (given the other cruft automatically compiled in, this was almost 3k). I changed the implementation to be a macro:

```
#define min3(x, y, z) (((x)<(y))?(((x)<(z))?(x):(z)):(((y)<(z))?(y):(z)))
```



Don't remember the ternary conditional operation? It is a shorthand version of an if-statement:

condition ? value if condition is true : value if condition is false

Not only does this make your code more dense, there are times that it nudges the compiler into more optimal code.

Then I modified my code to run the macro a few times to get different code space sizes. I changed the macro into a function and ran the same tests. It was a bit odd: the functions were the same whether I used an inline function, a regular local function, or an external function.



The `inline` keyword should have made the function behave as a macro did, replicating the code into each location. However, it is only a suggestion to the compiler, not a requirement. How the compiler takes the suggestion is very compiler specific.

After I recorded the size, I turned optimizations on and re-ran both tests, recording the difference in bytes each implementation was from the baseline. All along I had all my variables marked as volatile so that the compiler couldn't remove the interim function or macro calls.

Implementation	1 call (diff from baseline)	2 calls	3 calls
Macro	0	76	152
Function (local or external)	20	60	96
Macro with space optimization	-40	8	56
Function with space optimization	-40	-20	0

The key thing to note from the table is where the crossover points are (2 calls without opts, 1 call with). A macro uses the same amount of space as copying the line of code into your function (essentially the preprocessor does a search and replace). Once you've called the min3 macro a few times, it would take less code space to turn it into a function. Although the function call generally makes for smaller code, we'll have to return to the issue when we look at functions in RAM and processor cycle optimization.

Macros do have the advantage (of sorts) that they don't do type checking; the same macro could be used for integers, unsigned integers, or floating point numbers. Also, a smaller snippet of code may never have a crossover point, so a macro may always be better. (Even the innocuous minimum of two has a crossover point of three calls, so a smaller snippet of code should be very small indeed.) Crossover points depend on many factors including processor architecture, compiler implementation and memory layout. You may need to do some experimentation on your system to determine the code space trade off between macros and functions.

Constants and Strings

Going through the map file, you may find that your debug strings take up a lot of your code space. This is a really tough dilemma because those debug strings provide valuable information and useful comments.

If you've implemented a logging API (described in [“From Diagram to Architecture” on page 16](#)), you already have the ability to turn the debugging on and off per subsystem at runtime. To get rid of the debugging strings, you'll need to go a step further and remove the functions at compile time.

You can still get some flexibility (and some documentation) using this common idiom that lets you turn off the logging in a particular subsystem (i.e. the motor subsystem):

```
#define MOTOR_LOG 1 // set this to zero to turn off debugging
#if MOTOR_LOG
#define Log(level, str)          Log(eMotorSubSystem, (level), (str))
#define LogWithNum(level, str, num) LogWithNum(eMotorSubSystem, (level), (str), (num))
#else
#define Log(level, str)
#define LogWithNum(level, str)
#endif
```

When you change the value of `MOTOR_LOG` to 0 in the `#define`, all strings get compiled out (even though the code looks the same). It can be a little frustrating if you forget and

try to turn debugging on during runtime so you'll need to find the balance between runtime flexibility and code space.

As for other constants, why are they there? Do you really need them? Are there ways to calculate the data at runtime? Or compress it? Can you move the information to another storage mechanism (an external device such as a flash or EEPROM)? The options depend on your system, but now that you know where the space is going, you can dig into the problem.

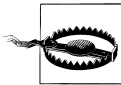
RAM

Unlike the angst produced when your linker says you are out of code space, a RAM resource error from the linker should give you a sense of relief at having dodged a bullet. The alternative symptom for a RAM shortage is a system that crashes randomly.

Some of the same techniques for finding more code space work for RAM as well. However, it's more difficult to find out where the RAM is disappearing to. You can make it easier on yourself with some design choices.

Free malloc

To really understand where your RAM goes, eliminate dynamic memory allocation. Local variables are hard to see, so if you make large arrays global, you can use the memory map output of your linker to determine where the RAM is used.



If your system uses a language with garbage collection, you might not have the luxury of knowing where your RAM is going. For embedded systems written in such languages, dealing with RAM constraints becomes much more difficult.

If I haven't convinced you that the transparency is a good enough reason to get rid of dynamic allocation, there are some other reasons. Say you have a heap that is 30 bytes in size (this is a little small, but makes my example easier) and in a particular function, you allocate a 10-byte buffer and a 5-byte buffer. Then you free the 10-byte buffer so you can allocate a 20 byte buffer. What is wrong with this picture? More than you might expect:

- Wasted RAM. The heap requires RAM to keep a data structure describing the memory that's in use. Every dynamic allocation has some amount of metadata overhead.
- Lost processor cycles. Keeping track of the heap is not free. Searching the heap data structure for available memory is usually a binary search, which is pretty fast, but it is still a search.

- **Fragmentation.** As shown in [Figure](#) , after you've allocated the 10-byte and 5-byte buffers, the first half of the heap is used. After you free the 10-byte buffer, there are 25 bytes available. However, there isn't a contiguous block for the 20 byte buffer. By mixing buffer sizes, the heap gets fragmented. With a heap that is much larger than the variables (and few or no small allocations), this is less of a problem. However, in an embedded system, you may not have the luxury of a large heap.

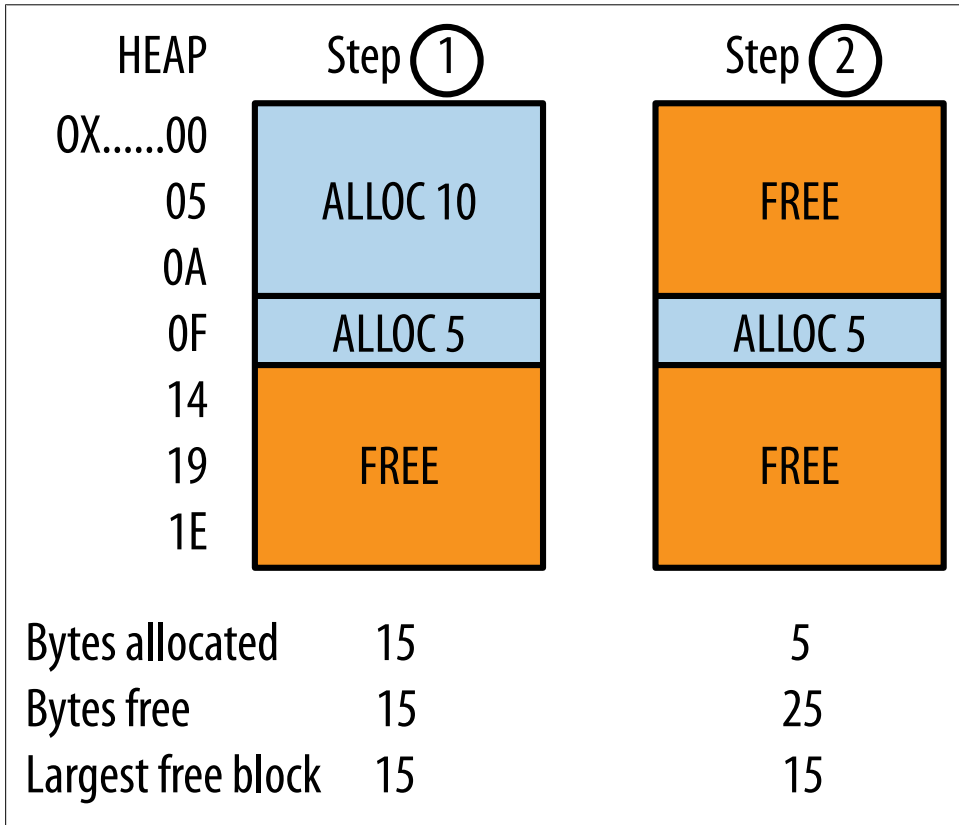


Figure . Heap Fragmentation

If you have a buffer that can be recycled after it is used, there are options besides using the built-in dynamic allocation system. If you need two buffers—say, one for an interrupt to write to and one for the normal code to read from—use a ping-pong buffer as described in [“Interrupts” on page 123](#). If you have a queue of data, implement a circular buffer ([Chapter 6](#)). If all of your buffers are the same size, consider a chunk allocator (there many examples on the web and Wikipedia). However, avoid reinventing the wheel. If your memory system is so complex that you end up rebuilding malloc, use the built-in version. Its benefits will probably outweigh the costs on your system.

Stacks (and heaps)

A stack is a data structure that holds information in a last-in-first-out (LIFO) manner as shown in [Figure 8-2](#). You push data on to 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).

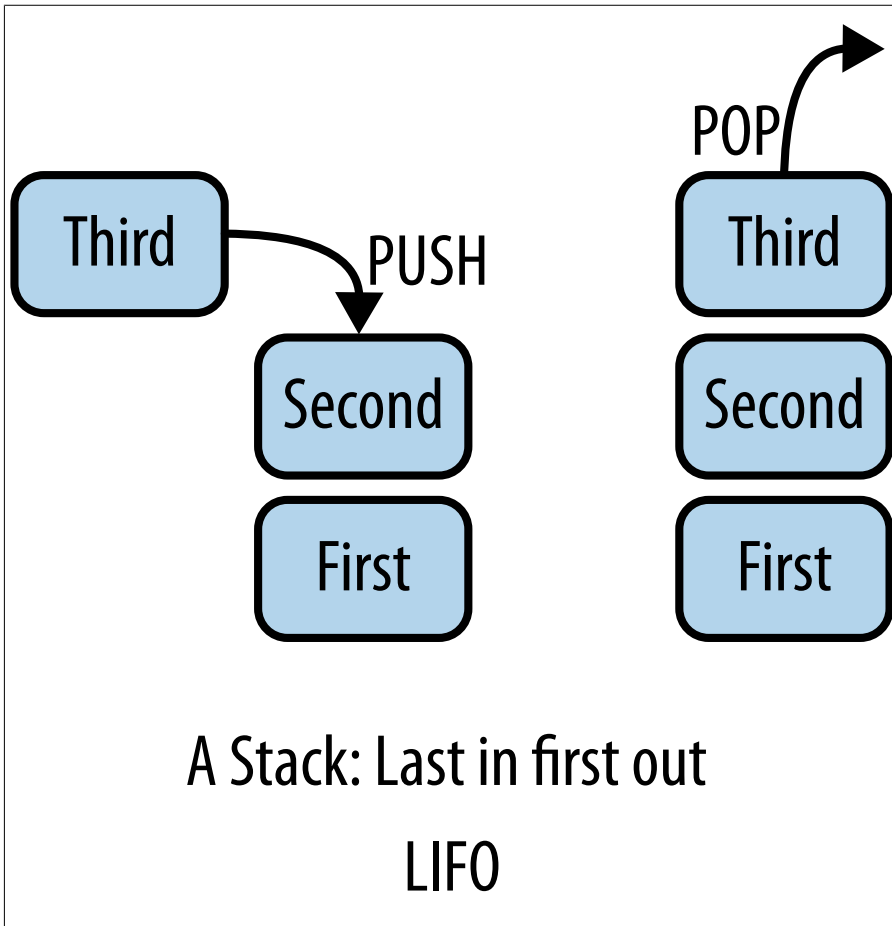


Figure 8-2. Stack basics

A stack is a simple data structure any student can implement, but *the* stack (note the definite article) refers to the call stack that lies behind every running program, 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, then whatever you call after that.

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).

The heap grows up (see [Figure 8-3](#)), whereas the stack grows down. When they meet, your system crashes. Well, actually your system will probably crash before they meet because there are other things between them (global and static variables and possibly code).

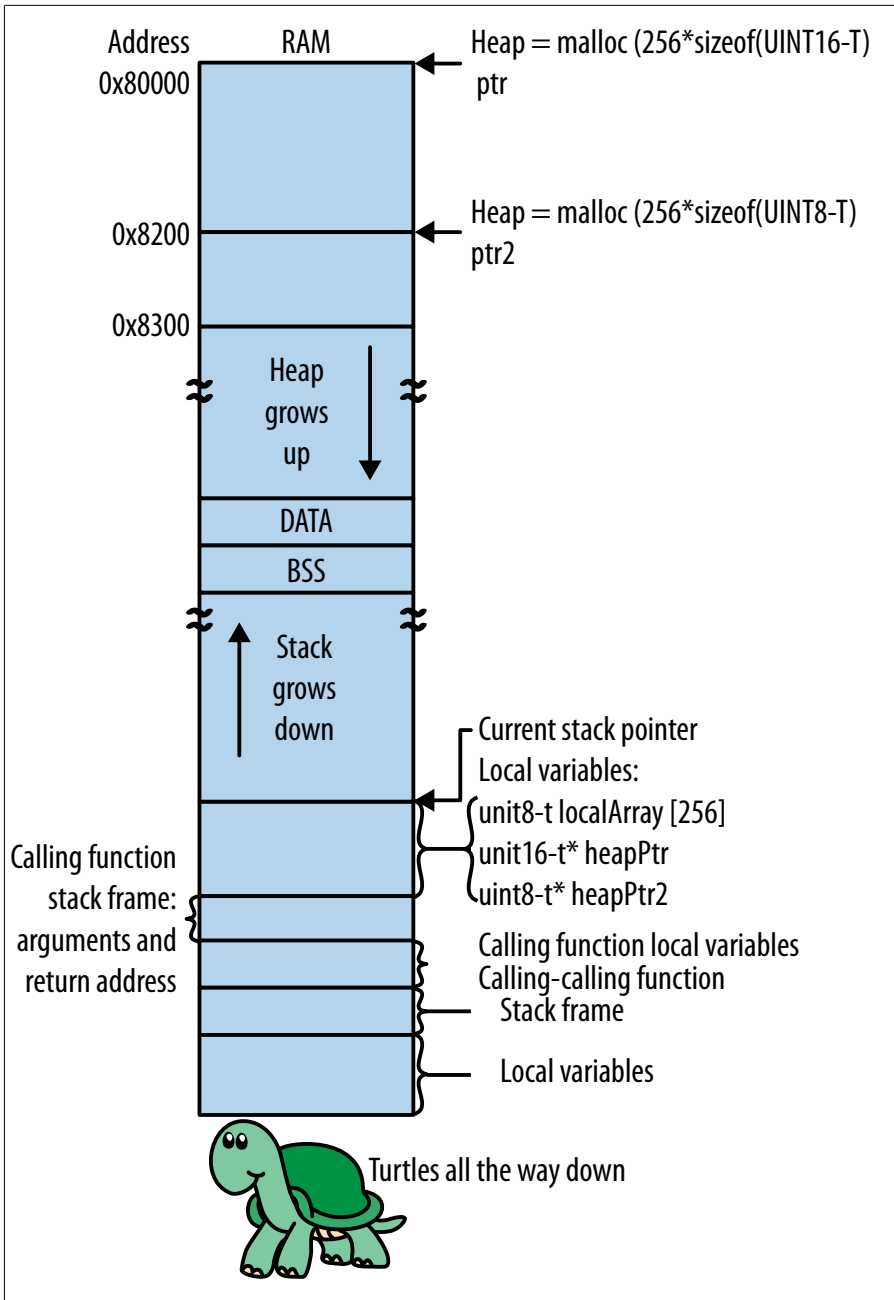


Figure 8-3. Memory map of a system

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. 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).

Long function call chains can make for a large stack, particularly if the intermediate calls have many local variables.



Recursion is seldom used in embedded systems because the stack may be exhausted before the solution is obtained.

Allocation for the heap and the stack is often done in the linker file (or as an argument passed into the linker). Without malloc, the heap can be allocated to zero (or only as large as your libraries require). However, the stack should be a bit larger than the amount you calculate (I'd say 25% larger). Stack overflow bugs are very difficult to solve, and a bit of buffering will save you from needing to do so. (If you have an operating system, each thread may have its own stack and heap.)

Reading a Map File (Part 2)

Going back to the map file, we may find some RAM information in the data and bss sections. They are not always obvious: you may need to search for these sections because they often look like the text section filled with functions. For each section there is a summary that shows the total amount used:

.data	0x10000000	0x144
...		
.bss	0x10000000	0x1b7c



In some map files, the data section is called cinit. Most linkers are pretty standard, but if you see things you don't recognize, you may need to break out the linker manual or search online.

Looking back to the overview memory map (the memory configuration in the code space section), this program is taking up quite a lot of its 0x2000 bytes of RAM. Note that this total doesn't include the heap or the stack. (Actually, there is no heap in this program, so all remaining RAM is allocated to the stack.)

Remember, the data section contains constants for your initialized variables and bss contains all of your uninitialized global and static variables. The data section variables also require code space for their initial values, which is captured in the rodata section.

The bss section variables are RAM only. The layout is similar to what it was for the functions.

.section	address	size	file
	address		global variable name
	address		global variable name

If your variables are local to your file (or static variables in a function), only the size and file name are shown.

.bss	0x10001bb4	0x14	./SharedSrc/i2c.o
------	------------	------	-------------------

As with functions, the goal here is to look for the larger variables, as those are where the best savings can be had. (Reducing a 12-byte array by 50% is not as exciting as doing the same for a 200-byte array.)

Registers and Local Variables

Unlike global variables, it is more difficult to estimate the RAM consumed by local variables (those within functions). Registers are the memory pieces of a CPU. Some processors can't perform actions on RAM directly. Instead, they have to load the value from RAM into a register, perform the action on the register, and then store the information back to RAM. That's a lot of instructions for a line of code that says:

i++

Not every processor needs to go through such gymnastics, but registers are faster for any processor to use. The reason local variables are difficult to count when tallying up RAM usages is that many local variables spend their whole existence as registers.



C has a keyword called `register` that is supposed to give a hint to the compiler about which variables you think should go in registers. The keyword is generally ignored, if it is implemented at all.

Function Parameters

A function's input arguments are often in registers instead of on the stack (RAM). You can encourage the compiler to do this if your function has a only few parameters per function. A good rule of thumb is less than four parameters per function. If you have a 32-bit processor, that would be four 32-bit variables, not a dozen pieces of data crammed into four structures.

In fact, if you have an N-bit processor, try to stick to N-bit variables. Larger variables generally are not candidates for precious registers. (Smaller variables often add processor steps. The compiler tries to access only the part you find interesting.)

To use registers, you usually want to pass arguments by value. If you pass by reference (or by pointer), you are sending the address of the data. The address may be in a register, but the data will need to be accessed in RAM. For example:

```
int bar = 10;
foo(&bar); /* this takes more RAM than passing by value
           because bar has to be in RAM to have an address */
```

There is an exception to the pointer guidelines if you pass a structure to a function. If you pass a structure by value, you are likely to have two copies of it in RAM: the original and the stack version of the copy. Pass pointers to structures to eliminate duplication.

Minimize Scope

For most efficient register use, each function should use only a small number of variables at any time. When optimization is on, your compiler will move around the code to limit the scope of the variables and free up registers. However, you can help it figure out what is going on. For a simple (and somewhat lame) example, take this code snippet that needs to make each array value equal to its place in the array. It does some processing and then needs the array to be reinitialized to its original values before going on:

```
for (i=0; i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
... /* do stuff to array, need to set it up again */
/* i still equals max array so subtract one and run through the loop again*/
for (i--; i >= 0; i--) { array[i] = i; }
```

The programmer thought they'd save a few cycles by not re-initializing `i`. However, as the code is written, its value has to be remembered across other processing even though it doesn't really matter. The variable's scope is increased without cause. We can rewrite the code to enable the compiler to forget about `i` when it isn't in use (and the compiler can reuse a register for a different variable).

```
for (i=0; i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
... /* do stuff to array, need to set it up again */
for (i=0; i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
```

For this example, the solution is a no-brainer (and has cleaner code). However, when you look at your code, can you see places to decrease the scope of the variables? Maybe do all the processing on one variable before shifting on to another?

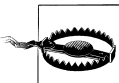
Don't worry about the total number of variables you have in a function. If you limit the scope of each variable and, within each scope, limit the total number of variables in play, the compiler can figure out the rest. It doesn't always pay to reuse variables, particularly if your compiler can't recognize an old variable used in a new scope.

Look at the Assembly

How do you know if you've been successful in your attempts to mind meld with the compiler? Look at the assembly code. No, wait, stop running away!

I'm not suggesting you learn assembly language for this. Well, maybe read it a little. I definitely am not suggesting you write in assembly language (that is in [“Coding in Assembly Language” on page 248](#) later in chapter). However, you can learn a lot about your compiler (and your code) by looking at the list file (`.lst`) or walking through the

assembly in your debugger. Make sure you can see the higher level language at the same time as the assembly. Your code acts as sort of a comment for the assembly code so you know what it is trying to do. (You may need to add a compiler option to retain the list files; look on the web or in your compiler manual.)



The first few times you look at assembly and walk through it, make sure optimization was off during compilation. The tricky things that occur during optimization are not a good introduction to learning how to read assembly.

As you look at the code in a new light, note that when the compiler optimizes for speed, it will put the most often used variables into registers so that it doesn't have to load and store them for each access. If you can set your compiler to optimize for RAM usage, it will instead put the largest number of variables in registers, possibly leaving an often used variable in RAM if its scope is large.

Function Chains

We saw in the “Stacks (and heaps)” on page 225 that each function call increases the size of the stack. Say we have three functions: `main` calls `foo` and `foo` calls `bar`. At that point, our stack looks like the left part of Figure 8-4.

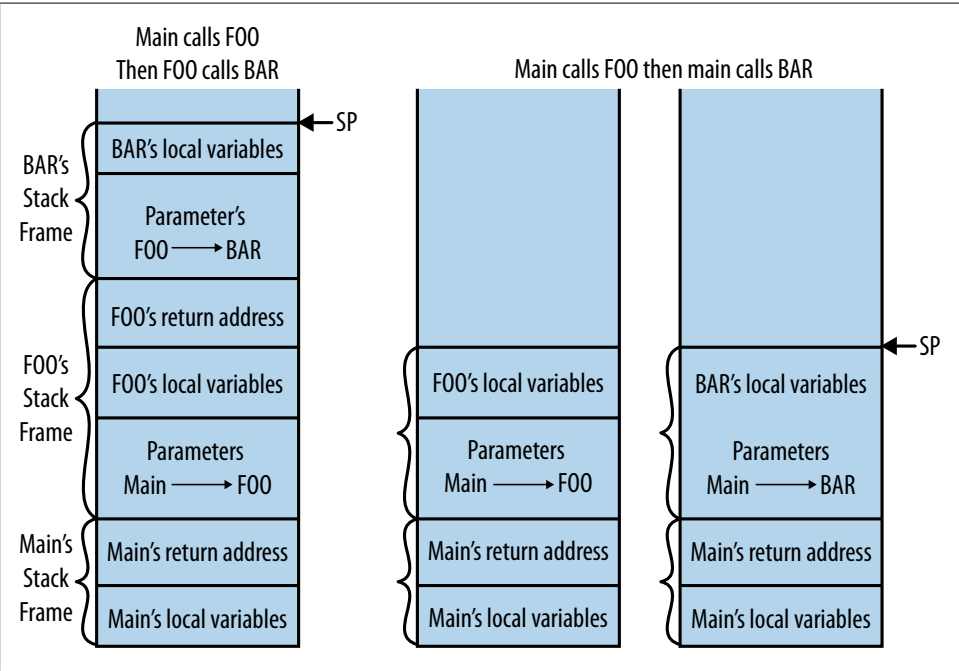


Figure 8-4. Function chains in the stack

Only a few registers are available on the processor, so everything else goes on the stack. You don't get extra registers for making function calls; there are a static number through the whole system. So, if you have a chain of functions, the local variables and parameters from earlier functions will have to go on the stack to make way for the latest calls. Even if a parameter or a local variable starts out as a register, if it has a large enough scope, it may end up on the stack when you call another function.

As shown in the right side of [Figure 8-4](#), if you can tweak your design to have a flatter function structure, your stack can be smaller (and each stack frame will be smaller as you increase the register usage).

One exception to the rule that functions calling functions incur RAM costs is a technique called *tail recursion*. It isn't really recursion (remember: recursion bad in an environment with limited RAM!). In tail recursion, you call the next function as the last statement of the current one (for instance, call `bar` at the very end of `foo`). This lets you retain encapsulation for your modules *and* keep the stack small. The compiler will remove `foo`'s local variables and parameters from the stack, allowing the `bar` function to return directly to `main` (without passing through `foo`). Even though `foo` called `bar`, the stack looks like the right side of [Figure 8-4](#).

Also, while an earlier section discussed the trade-offs of macros in terms of code size, macros play a different role in RAM reduction. Macros take up no stack frame at all, so they are good for decreasing the RAM footprint (especially for little functions like “[Functions and Macros](#)” on page 221). A little more code space might lead to less RAM (and faster execution).

Pros and Cons of Globals

Global variables have a bad reputation. They've been party to all sorts of spaghetti code. They continually conspire with inexplicable outside forces to control the flow of code. They don't know the meaning of the word *reentrant*.*

But laziness convenience is not the only reason for using global variables. They do have a good side.

But a little more about their bad side first. Global variables can't be stored in registers. That means they always take RAM. They are almost always slower to access than a local (register) variable. So if you've got a global variable that could be used as a local one, make it local. Even if it ends up on the stack instead of in a register, once the program execution goes out of scope, that RAM is free for use elsewhere.

When can a global save on RAM? Say you have a function chain and need a particular variable in several functions (or worse, you only need the data at the end of your function chain). If you pass the data in via a parameter to each function in the chain, with

* A subroutine is *reentrant* if it can be interrupted in the middle and safely be called again before its previous invocation has been completed.

intermediate processing, the compiler may be unable to keep the data in a register all the way through the chain. Because it is a parameter to each function, the data will go on the stack multiple times. A global variable is outside the stack and short-circuits the process.

Memory Overlays

Does your system have a few large buffers? Some of the particularly bulky ones I've found on some of my systems include display buffers, communications buffers, and sensor input buffers. Once you've identified the large buffers in the system, ask yourself whether they all need to be used at the same time.

For example, maybe your display buffer is used to build up the image and then sits idle until the next time you need to update the display, at which point you build it up again. Or the sensor input buffer isn't needed while you're waiting for an interesting event. Once you notice the event, you need the whole buffer to queue up the data, but until then, only a small area of it is needed. Or maybe you have a large communications buffer for when you are receiving new code to load, but in general you need only a little of it. Does any of that sound familiar?

If your system used dynamic memory allocation, these subsystems might allocate and free their memory to avoid tying up the RAM resources. However, if you've banned dynamic memory allocation (and if a chunk allocator is overkill), your system might benefit from RAM overlays. This is where two subsystems share some or all of their allocated memory, but only one gets to use it at a time. By overloading the resource, you effectively get a lot more RAM. [Figure 8-5](#) shows a 4k RAM buffer available on a processor. Without the overlay, they overflow the available space and the system won't compile. With the overlay, the buffers fit. However, the two subsystems depend on each other in a way not obvious to the casual observer (since you've smashed encapsulation to bits).

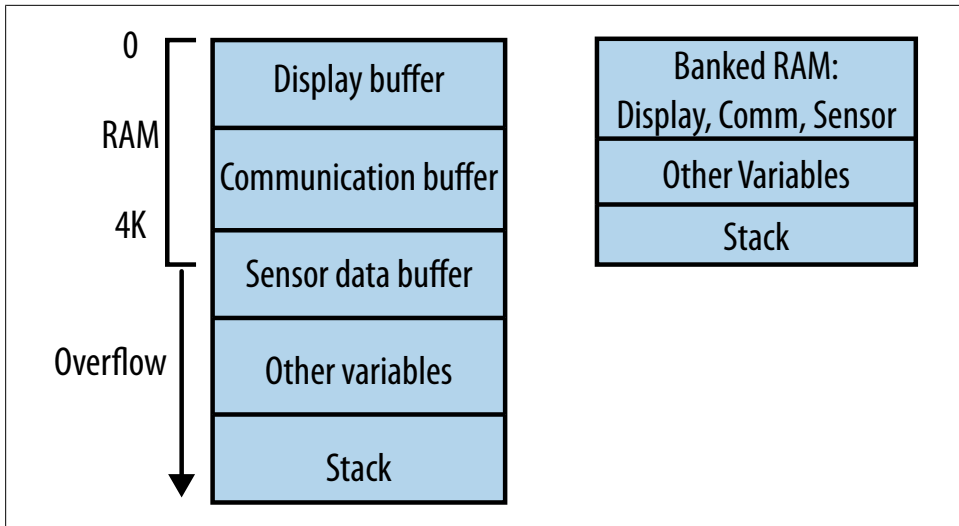


Figure 8-5. Sharing RAM between modules

There are several ways to implement memory overlays. The simplest is to treat the buffer as private to one module and allow access to it through a single function, as described with the modified singleton pattern in [“From Diagram to Architecture” on page 16](#). Another method is to implement a union of several arrays, possibly with an owner tag to indicate the current user of the memory.

Alternatively, to make the subsystems less entwined, you can modify the linker script to overlay the RAM buffers on each other. This eliminates any direct interaction between the two subsystems, but requires that any future developer understand that the two subsystems cannot run at the same time (without catastrophic results).

Speed

Of all the places to spend your valuable development time, the worst can be sunk in trying to squeeze out a few more cycles from the processor. Try to avoid that position by starting with the design of the system, maybe building in some overhead when selecting a processor. However, there are times when you have to make the code go faster to respond to a real-time event, to add another feature, or to reduce power consumption (see [Chapter 10](#)).

Before delving into serious system tuning, start by profiling your application to make sure you focus on the important parts. While not as simple as reading the map file, it is still unwise to optimize the wrong thing (e.g. cutting half of your initialization time, which runs only once at system boot, instead of focusing on the problem in your main loop).

Profiling

To know where to spend your time optimizing the code, you need to know where the cycles are going. Many compilers (and operating systems) have some built in analysis tools, including profilers. If you have those, learn them. If you don't, you may want to build your own rudimentary profiler to give you some insight.

In physics, the Heisenberg Uncertainty Principle says that electron momentum and position cannot simultaneously be known to an arbitrary precision. In profiling there is also uncertainty: the more precisely you need to know how long each piece of code takes to run, the less you can know about the whole of the execution. That is, your profiler will change the behavior (and timing) of the code. Understanding the impact your profiler has on your code is an important part of profiling.

Profiling usually starts off trying to answer questions about where the processor time is spent. By digging down into a particular function, you can apply the same method to determine which part of that function is longest (and so on). Each of the four profilers discussed here has a slightly different focus to help you get a broad and deep view of your system.



Similar to the code space and RAM score cards, use your profiler to track which modifications were most effective.

I/O Lines and an OScope

If you've got a few open I/O lines, they can show you where to start on the path to profiling your code. As you enter a function of interest, set an output line to be high. When you leave, set it low. Watch these lines on an oscilloscope to see how long each function takes.

For example, we have a system that waits for data to be ready, reads the data, transforms it and displays the result to an LCD:

```
Interrupt sets data ready variable when data is available to be read
```

```
Main loop:
```

```
Loop, waiting for data ready to be set  
Read the data in the buffer  
Transform data into information  
Write the information to the LCD
```

If we had four I/O lines, setting and clearing one for each stage, the result might look like [Figure 8-6](#). Each I/O switches exactly as its predecessor completes (often you don't need to instrument everything; you can look at the gaps). In the beginning, it looks like everything is good: the bulk of the time is spent waiting for the data to be ready and writing information to the LCD. Eventually, the wait for data ready goes to zero but the other tasks take up the same amount of time.

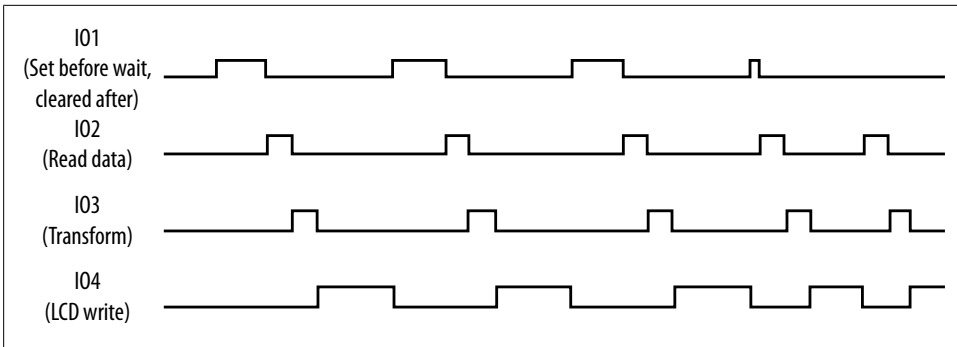


Figure 8-6. Profiling Using Oscilloscope Lines

Two things in the graph are important. First, as time goes on, the data is always ready as soon as the LCD write is finished. This means that you will start to miss data because the system can't keep up. Second, the system time is dominated by the LCD write, the activity on the bottom. If you can fix what is taking so long in there, the system might be able to keep up with the input.

The great part about I/O line profiling is being able to set and clear an output line in an interrupt. It will slow your system down to add the I/O changes to interrupt service routines, but only by a little. The profiling information you gain about the system is often worth the temporary slow-down.

Timer Profiler 1 (Function timer)

For a system with few interrupts, another way to implement a profiler is to time how long a section of code takes to run. In [“System Tick” on page 138](#), I talked about how to build a system clock to measure how long things will take. We built the `TimeNow` function to return the number of milliseconds since the processor started. Using that as a simple timer profiler might look like:

```
struct sProfile {
    uint32_t count;
    tTime    sum;
    tTime    start;
    tTime    end;
};
void main()
{
    struct sProfile profile;
    ...

    profile.count = 0;
    profile.sum = 0;
    while (1) {
        profile.start = TimeNow();
        ImportantFunction();
        profile.end = TimeNow();
```



```

    profile.sum += profile.end - profile.start;
    profile.count++;
    if (profile.count == PROFILE_COUNT_PRINT) {
        LogWithNum(eProfilerSystem, eDebug, "Important Function profile: ", profile.sum);
        profile.count = 0;
        profile.sum = 0;
    }
    ... // continue with other main loop functions
}
}

```

The function profiled (`ImportantFunction`) should be longer than the timer tick—at a minimum, twice as long, and preferably at least ten times as long. It is also critical that the profiling function (`TimeNow`) take a negligible amount of processing compared to the profiled function.

Note that the summation and logging are outside the scope of the profiler. Try to sample only the things you care about and not all of the accoutrements of profiling. If you aren't sure that you've eliminated the profiler overhead, try commenting out the function of interest so that you are profiling only the start and end time acquisition. The result should consistently be zero if the profiler is working properly.



If you have a few functions of interest as you try to decide where to spend your development time, you might use three or four profiler variables to monitor different areas of code.

Finally, using many samples will average out any minor differences (for example, if you have a short intermittent interrupt). By sampling multiple times, you also get a small increase in precision as the integers get larger. For example, with one sample, the function may take 10 ms. But with a thousand samples, the sum may give you 10435 ms, indicating an average of 10.4 ms.

[Figure 8-7](#) shows how each this function timer profiler measures one area of interest at a time.

Timer Profiler 2 (Response Timer)

If you want to profile a function that is much shorter than your timer tick, you can use a shorter timer tick in your profiler or change the way you sample. The following example, for instance, profiles the whole main loop, because the profiler timer doesn't stop until the counter has reached its goal as shown in the middle of [Figure 8-7](#). This may yield valuable information if you are trying to react to an event within a certain amount of time.

```

profile.count = 0;
profile.start = TimeNow();
while (1) {
    ImportantFunction();
}

```

```

    profile.count++;
    if (profile.count == PROFILE_COUNT_PRINT) {
        profile.end = TimeNow();
        profile.sum = profile.end - profile.start;
        LogWithNum(eProfilerSystem, eDebug, "Important Function profile: ", profile.sum);
        profile.count = 0;
        profile.start = TimeNow();
    }
    ... // other main loop functions are also part of the profile
}

```

This timer profiler has a larger impact on the results, because it increments the counter and compares it to a constant inside the profiled area. If your loop is very quick, these activities may be non-negligible. As before, you can check this assumption by commenting out everything in the main loop except the profiler. By taking this measurement, the baseline of the profiler can be subtracted out of your final analysis.

As shown in [Figure 8-7](#), this profiler gives you a longer view of the system timing. Both of the timer profiling methods break up the flow of code. They tend to be temporary pieces of code that you remove after gathering the information you want. Instrumenting the code in this fashion is straightforward, so reproducing the code as needed is usually better than leaving it in to slow down the code and/or confuse future generations.

Sampling Profiler

If you have an interrupt-driven system, profiling can be more difficult. However, if you can allocate a block of RAM to it, you can implement a sampling profiler with a timer-based interrupt.

First create a timer interrupt, one that is asynchronous to everything else in the system. For example, if you have 10Hz and 15Hz interrupts, make sure your new timer is not 1, 2, 3, or 5Hz. Instead make it something like 1.7Hz so it is not evenly divisible into any of your other time based interrupts. This makes sure that your results are not biased by periodic functions. [Figure 8-7](#) illustrates the relative timing of two functions and your custom timer (the little arrows at the bottom).

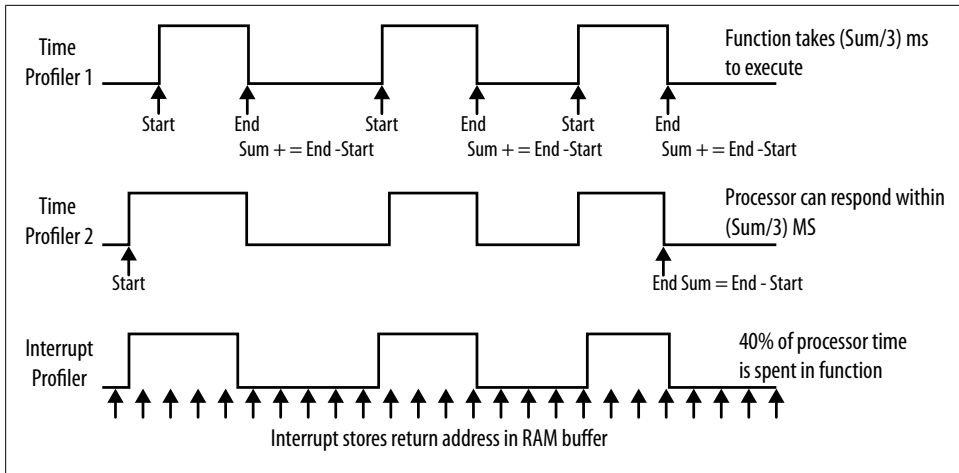


Figure 8-7. Comparing timer and interrupt profilers

Now, on every profiler timer interrupt, save the return pointer to the block of RAM. The return pointer tells you what code was running when the timer interrupted. Once the RAM buffer is full, stop the timer and output the list of addresses. Armed with a list of return addresses, figure out where these are in the image using the map file. A scripting language will significantly help you parse the map file and count up the number of times the profiler sampled from each function. You can thus figure out what percent of the processor time each function is taking.

This method works best when your processor allows nested interrupts and the profiler timer is the only one allowed to interrupt other interrupts. If you have other non-maskable interrupts, you won't be able to see those in the results.

This sampling profiler doesn't slow down any particular function, but it does place a tiny drain on the whole system. (Outputting the list of addresses may be a larger drain, depending on your implementation.) This sort of profiler is readily left in the code with the timer (and output function) turned off when it is unneeded.

Optimizing

Armed with the knowledge of what is slowing down your system, you can now explore the options available to you. Definitely start by turning on optimizations in your compiler. The less tweaking you need to do, the better your code will be.

Next, try to get most of your variables into registers. Even if you have a long function call chain and variables end up going on the stack while another function is called, if they can at least be registers in a smaller scope, the code will be faster.

Having done those basics, consider the techniques in the following sections.

Memory Timing

Wait states are a bane to efficient use of the processor resources. Many types of memory cannot be accessed as fast as the processor runs. To get information from such memory, the processor has to wait some number of processor cycles to offset the timing difference. Memory has a number of *wait states*. For example, if your code runs from four wait state flash, every time it needs a new instruction, the processor has to wait for four clock cycles.

I should point out that four wait state memory doesn't (always) mean your code runs four times slower than in one wait state memory. Processors can pipeline instructions to reduce the impact of slow memory: this means looking up the next instructions while executing the current one. If the number of pipeline stages is greater than the number of wait states, the memory will slow execution only when the pipeline stalls (which tends to happen at branches, such as `if` statements, and function calls, because the processor can't just assume that it will execute the next instruction in memory).

By knowing how many wait states each type of memory has, you have options for speeding up execution by moving code to faster memory. For example, you may want to copy a critical function to zero wait state RAM if that is faster than your normal program flash. If you need the function occasionally, you can overlay the memory with another buffer. When the function is needed, copy it out of code space (though you'll need to do some profiling to make sure the overhead of copying it is balanced by its amazing speed in RAM). Many compilers support a keyword (`ramfunc`), pragma or macro that lets you indicate the function should be stored in code space but loaded into RAM on boot (by the startup code that runs before `main`). Often, you'll need to modify your linker file to put the section into RAM. Check your seldom used compiler manual for this, not your handy processor manual.

Variable Size

When you are working on an 8-bit processor, it is easy to understand that using a larger variable will incur a higher cost in terms of processor cycles. What may not seem as logical is that using a smaller variable on a larger processor may have some unwanted overhead as well. For best results, local variables should match the size of the registers.

When the compiler needs to reduce the size of a local variable from a native 32-bit variable to a 8-bit `char` or a 16-bit `short`, it has to extend the sign, which means every manipulation takes two instructions (or zero extend for unsigned variables, which only takes an extra instruction). Cutting out one instruction at a time is a good way to drive you crazy. However, if you generally implement variables using native types, you won't have to sort through your code later.

Along the same lines, try to use the same type of variables for the bulk of your processing. Type conversions are a waste of processor time. Converting between signed and unsigned should also be avoided as much as possible.



Signed ints are upgraded to unsigned ints when the two are compared. So a small negative signed variable is likely to be considered a numerically larger value when unwisely compared with an unsigned variables. Stick with unsigned variables unless you really need negative numbers.

One Last Look at Function Chains

Functions make your code run more slowly, because the processor switches context by pushing data on the stack (and usually having to refill the pipeline). Of course, functions make your code maintainable (and usually make it more correct), so don't eliminate them entirely. Try to avoid small functions where the cost of calling the function may outweigh its benefit.

For example, take an LCD display driver. LCD drivers are often optimized because the data needs to be on the screen before the screen refreshes. If there is no other form of control and only part of the data is available in the LCD's buffer, the screen may tear (show part of the old image and part of the new) until the next refresh cycle. Our example driver is for an LCD that is 240x320 pixels and each pixel is 16-bit (meaning it has 2^{16} colors). When putting an image on the screen (or part of the screen), the driver reads from a buffer of RAM and puts it on a parallel bus that is 8-bits wide. The code starts out looking like:

```
// in Lcd.c
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength)
{
    int i;
    while (bufLength) {
        LcdWriteBus(buffer[i] & 0xFF);      // write lower byte
        LcdWriteBus((buffer[i] >> 8) & 0xFF); // write upper byte
        i++;
        bufLength--;
    }
}

void LcdWriteBus(uint8_t data)
{
    IoClear(LCD_SELECT_N);      // select the chip
    IoWriteBusByte(LCD_BUS, data); // write to the IO lines
    IoSet(LCD_SELECT_N)        // deselect the chip
}

// in Io.c
IoWriteBusByte(uint32_t io, uint8_t data)
{
    // ioBus was configured during initialization
    ioBus[io] = data;
}
```

That is a lot of code, and when it's lined up like that you may see some chances for optimization. When the code is scattered in multiple files, it can be harder to see.



When you find a spot that seems worth optimizing, look at its call chain as a whole.

Selecting and deselecting the LCD for every write is silly. The `LcdWriteBus` function has to do it because it doesn't know what else is going on around it. However, the `LcdWriteBuffer` function knows it is sending a large amount of data. While the `LcdWriteBus` function is probably used by other parts of the code, the `LcdWriteBuffer` code won't call it, instead implementing its own version.

```
//in lcd.c
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength)
{
    int i;
    IoClear(LCD_SELECT_N);          // select the chip
    while (bufLength) {
        IoWriteBusByte(LCD_BUS, buffer[i] & 0xFF); // write lower byte
        IoWriteBusByte(LCD_BUS, (buffer[i] >> 8) & 0xFF); // write upper byte
        i++;
        bufLength--;
    }
    IoSet(LCD_SELECT_N)            // deselect the chip
}
```

So you've eliminated one link in the function chain. The function will run faster and there is no way the compiler could have done that optimization for you. All you had to do was copy code from one function to another, not sell your soul or anything. However, if there was a bug in `LcdWriteBus`, it is likely to now be in `LcdWriteBuffer`. Maybe you should add a comment in both functions to cross reference so a maintainer will know to change both locations.

The slope gets much more slippery from here. The next function in the chain is `IoWriteBusByte`. It is only one line of C code, but underneath it does some indirect addressing to write to the I/O register. It does that every time the function is called even though the byte is written to the same LCD I/O lines each time.

It seems like an easy change. However, the I/O function is in the I/O module, not the LCD module. If you change this, your code becomes faster but less portable, as it won't have any abstraction between the LCD and the hardware. Is it worth it? Are you looking at a situation where you need 20% more cycles or one where you absolutely need 60% more cycles?

The cost of making a dramatic improvement is often high. Even if the change is fast to make now, future development time will need to be invested as the system becomes less flexible and more information intensive. The next person is not going to recognize the trade offs, so they will stumble over the code and ask why you made it so ugly.

There is another option. By making `IoWriteBusByte` a macro, you can preserve the modular boundaries and still eliminate the stack manipulation. The macro will take up a little more code space (especially if `IoWriteBusByte` is used in many other functions). Worse, you still are still spending processor cycles with the pointer access. However, before we think about breaking encapsulation further, let's think through some other options.

Consider the Instructions

If you were the processor, what would go on with the instructions that appear in the two `IoWriteBusByte` calls, to read `buffer[i]` twice and then increment `i`? Though the machine code is processor dependent, we can estimate the steps for each line of code (in italics):

```
IoWriteBusByte(LCD_BUS, buffer[i] & 0xFF); // write lower byte
(Variable i would already be in a register, already initialized)
Copy the buffer pointer from the stack into a register
Add the buffer pointer to i
Read the contents of memory at that address
Perform bit-wise AND with contents
Put i on stack
Call IoWriteBusByte, passing data
Pop i off the stack

IoWriteBusByte(LCD_BUS, (buffer[i] >> 8) & 0xFF); // write upper byte
Copy the buffer pointer from the stack in a register
Add the buffer pointer to i
Read the contents of memory at that address
Perform shift with contents
Perform bit-wise AND with result
Put i on stack
Call IoWriteBusByte, passing data
Pop i off the stack

i++;
Increment register i
```

I broke the process down into pretty granular steps. Although some assembly languages could combine some of my instructions, most of them would need a few more. The goal is to think like a microprocessor. And, if you do this, even if you are wrong sometimes, you will write better and more easily optimized code. Thinking along those lines, it would be more efficient if we didn't have to add the index to the buffer pointer.

Making a PB&J

There is a neat experiment you can run if you have a handy 5-9 year old. Ask them to pretend that they have a robot (you) and want to make a peanut butter and jelly sandwich (PB&J). The 'bot is very dumb so they have to give it complete and detailed instructions.

The robot already knows how to grasp and lift objects (bread, peanut butter, jelly, bread and spreaders). But it doesn't know how to put it all together to make a sandwich. What are the steps and how do they go together? As the child writes (or says) the steps, the robot often creates a mess instead of a sandwich as it interprets each command literally.

Breaking down the process of making a PB&J is a fun way to get kids to understand a little bit about the process of writing software. Plus, you get to say “Does not compute” like a Dalek until they manage a sandwich.

(Thanks to Walter and Emma Stockwell for letting me experiment upon their sons, Alasdair and Toby.)

```
IoWriteBusByte(LCD_BUS, *buffer & 0xFF); // write lower byte
```

(Variable buffer would already be in a register, already initialized)

Copy the buffer pointer from the stack in a register

Add the buffer pointer to i

Read the contents of memory at that address

Perform bit-wise AND with contents

Put buffer on stack

Call IoWriteBusByte, passing data

Pop buffer off the stack

```
IoWriteBusByte(LCD_BUS, (*buffer >> 8) & 0xFF); // write upper byte
```

Copy the buffer pointer from the stack in a register

Add the buffer pointer to i

Read the contents of memory at that address

Perform shift with contents

Perform bit-wise AND with result

Put buffer on stack

Call IoWriteBusByte, passing data

Pop buffer off the stack

```
buffer++;
```

Increment register buffer

```
i++;
```

Increment register i

This doesn't really change anything important; it only gives a hint to the compiler about what choices it could make to go faster. (And some compilers are intelligent enough to have parsed the code to get to the same point.)

The immediate lesson is use pointer arithmetic (`buffer++`) instead of arrays and indexes where possible. Pointer arithmetic uses a little less RAM and give the compiler a clearer path toward optimization. The larger lesson is to understand how your code is translated into machine language, and to make it easy for a compiler to take a faster route.

Reduce Math in Loops

What next? Ideally, we'd like to do less math during a loop. The shift and bitwise-and are relatively cheap, but if you do them on every pass, they add up. It wouldn't be necessary if the buffer was a byte buffer instead of a word buffer:

```
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength)
{
    uint8_t *byteBuffer = (uint8_t *) buffer;
    // buffer is now a buffer of bytes.
    // This works only if your endian-ness matches what the hardware expects
    bufLength = bufLength*2;

    IoClear(LCD_SELECT_N);          // select the chip
    while (bufLength) {
        IoWriteBusByte(LCD_BUS, *byteBuffer);
        bufLength--; byteBuffer++;
    }
    IoSet(LCD_SELECT_N)            // deselect the chip
}
```

The code now relies on a feature of the processor (endianness) that will not be portable. However, it will make it faster—probably. Remember that memory size matters. This change would improve the speed of an 8-bit chip significantly. For a 32-bit chip? Well, it shouldn't slow it down much because we were already doing the bit manipulation previously shown. At least now we are letting the compiler do the best optimization it can. The potential improvement depends on your architecture.

Usually, making the loop smaller is a good idea, as is making it do exactly the same set of operations each time. Keep `if` statements out of loops, as they mess with your pipelining and slow it all down.

However, there is a conditional statement in the previous loop. At this point, it may be taking a non-trivial number of cycles. If you were to walk through the assembly code, you might see that the loop is writing the byte, decrementing and incrementing some registers, and checking to make sure one register is not zero.



Checking against zero is cheaper than checking against a constant (much cheaper than checking against a variable). Make your loop indexes count down. It saves only an instruction or two but it is good practice.

Loop Unrolling

By now, when you look at the assembly for the loop in this function, the check against zero may be one of about ten instructions. The decrement of `bufLength` is another. You are spending 20% of your time dealing with loop overhead.

The compiler can't get rid of that, but you know more about the situation. For example, you know that there are an even number of bytes (because you made it that way). There is no need to check the loop on every pass, only every other pass:

```
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength)
{
    // Use a buffer of bytes. This only works if your endian-ness is correct
    uint8_t *byteBuffer = (uint8_t *) buffer;

    IoClear(LCD_SELECT_N);          // select the chip
    while (bufLength) {
        IoWriteBusByte(LCD_BUS, *byteBuffer);
        byteBuffer++;
        IoWriteBusByte(LCD_BUS, *byteBuffer);
        byteBuffer++;
        bufLength--;
    }
    IoSet(LCD_SELECT_N)             // deselect the chip
}
```

Now, if this loop previously took ten instructions to write a 16-bit pixel, it will instead run in eight. We've eliminated an increment to `bufLength` and a check against zero for a 20% improvement (yay us!). Loop unrolling, then, means reducing the number of iterations by duplicating code inside a loop. Improvements that large are not usually so cheap. Now we look back to `IoWriteBusByte` as the largest consumer of cycles in the loop, even as a macro. We'll stop here because, as noted above, it accesses variables in another file so it is a modification more egregious to maintaining modularity.

Why don't I care about removing the `bufLength` multiplication at the top of the function? For the same reason I never mentioned breaking modularity to reduce `IoClear` and `IoSet`. Reducing the code outside the loop is much, much less important because it runs only once (or once per function call).

When you are optimizing, focus on the loops and the repeated actions.

When I introduced the LCD function, I mentioned the screen was 320x240 pixels. Could you have a single function write out a whole screen's worth of data without any conditionals? Of course. And it would be faster than having conditionals. However, you'll be sad when you've lost count after pasting the 12243th instance of:

```
IoWriteBusByte(LCD_BUS, *byteBuffer); byteBuffer++;
```

Worse, when you compile, you may find that all those repeated instruction come with a cost: code space.

Sidebar: Explaining optimization to a business major

Ultimately, optimization is an economics problem. You start out with a portfolio of assets associated with your system. The major ones are the RAM, processor cycles, and code space. You may also have some auxiliary assets in power consumption and peripheral support on your processor. Your most liquid asset is development time, so

think of that as money. Once you invest (or allocate) these assets to parts of the system, you lose the opportunity to use them in other subsystems.

If the set of investments you initially selected do not deliver the expected returns, you might partially recoup the spent resources, but to do so you will need to sink more development time into the project. As you go further along the development path, switching the allocations becomes more and more costly (and less and less possible). Making the right call at the start of the design process can mean the difference between buying at the bottom of the market and losing your shirt.

In order to make strategic trades that are truly beneficial, you need to understand your resources in terms of the assets and debts. Unfortunately, the system portfolio is not yet a slick prepared prospectus that comes in the mail. An accurate, precise understanding of it takes a lot of time (aka money). Estimation can suffice for certain areas, others require in-depth investigation (profiling).

Your system's assets are somewhat fungible. For instance, you can trade RAM for processor cycles or code space (you will need some development time to facilitate the transaction). Sometimes, you don't have to trade anything; spending more development time will reduce your debt load. However, the total amount of any resource is fixed and a resource can become so scarce that any amount of development time (and assets) will fail to produce any more of it.

As you consider different optimization strategies, consider as well the high opportunity cost that comes with investing your assets. Buying futures is almost always cheaper than buying stock. Similarly, selecting resources at the design phase is cheaper than waiting until the last minute and realizing you are seriously in debt with no easy way to recover.

(Thanks to Jane Muschenetz for helping with the business side of this analogy!)

Lookup Tables

Nothing is ever free in optimization. Even turning on the compiler optimizations takes development time, because the debugger no longer works as it did: when you step through line by line, the debugger won't show you most of the variables (the ones the compiler removed) and it jumps around (because the compiler rearranged the assembly instructions).

Loop unrolling trades code space for speed. Using a macro instead of a function also trades code space for speed. These trades go either way, so if you need more code space, you may want to roll up any loops and eliminate macros.

There is another trade between cycles and code space (or RAM) that goes beyond macros: lookup tables. When the running code needs the information, instead of performing calculations, it finds result in the table, often with a simple index.

For simple example, if you need to transform from the 128 ASCII characters into their Unicode equivalent, you can put them in a table to look up the result. Lookup tables

go far beyond this though, many complex algorithms (i.e. encryption's AES) have two implementations: a fast one (large) one that uses lookup tables and a small (slow) one that computes the values it needs. (Generating lookup tables to reduce math overhead is covered in gory detail in [Chapter 9](#).)

The lookup table can be code based (where the values are calculated at or before compilation) or RAM-based. The latter are useful if your RAM access is fast (fewer wait states) or you have plenty of RAM but not enough code space (calculate the table on initialization). They can also be useful if your lookup values need to change when circumstances change (e.g. a networking router's lookup tables change when a different protocol is needed).



The compiler may create a lookup table when you use a switch statement in place of an if-else. Looking at your assembly code will tell you if it has (and if it is a savings to you).

Coding in Assembly Language

Looking at your assembly code is an important part of optimizing. Even if you don't start out knowing the meaning of the instructions in your assembly language, following along in the debugger for critical functions will help you figure out what the processor is really doing. Once you understand that, you may be able to tweak your high level language code to help the compiler do something more efficient.

That said, your time is valuable. *Don't program in assembly language*. Really, it isn't a good idea. The resulting code is usually a mess, almost always easier to rewrite than it is to read, even by the original programmer. That isn't to say it is easy to rewrite, just that it can be impossible to read. Maintenance to assembly code tends to be in the form of major refactoring (aka rewriting it all, fixing the old bugs but indubitably writing new ones).

However, if you have a function that really, really needs to be super-fast and the compiler is clearly not doing all it can, well, programming in assembly can be kind of fun, in a furtive playing-Tetris-at-work sort of way.

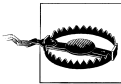
Start off with the high level code. Understand the assembly generated by the compiler. Next, turn optimizations on and really understand the new assembly generated by the compiler. You are smarter than the compiler and you have more system knowledge. But the compiler almost certainly knows the assembly language better than you do. Let it show you what it thinks is important before you start tweaking the assembly that the compiler output.

That is the big secret: don't start with a blank slate. Start with the assembly your compiler gives you and go from there. Paste the equivalent high-level code into the assembly code as comments so that your future-self can remember the goal of each section.

Summary

I can't provide a comprehensive list of optimization techniques. They depend on your processor and your compiler. Even more importantly, they depend on your requirements and your available resources. I can say that if you've gotten to the end of the project and you are looking for cheap ways to reduce your code by 40%, you have seriously misjudged the ability of your hardware resources to support your application. But I've made that mistake too.

Optimization starts with a good design, preferably one with a little extra overhead just in case the future features turn out to be larger, slower, and more RAM intensive than you'd planned. Even at the end of the project, you need a few spare resources to fix bugs and respond to hardware changes.



In medical and safety-related products, that margin needs to be larger. Recompiling the code is a paperwork-intensive process even for a minor one-line change. If you end up changing 50 lines because you need one more byte of RAM or just a few more processor cycles to implement the change, the paperwork and testing grows exponentially.

While there is some fun to be had in reducing the resources as far as possible, you'll get diminishing returns. While the first 10% of improvement shouldn't be too difficult, the next 10% will probably take twice as long (and so on). Start with a goal and don't optimize much beyond there.

There are trade-offs in optimization. You can share resources between subsystems. But every time you do that, your product becomes a little less robust and a little more fragile for the next person to modify. There is something incredibly embarrassing about handing a project off to another team and explaining why they should never change subsystem X because it will change the very precise timing of subsystem Y.

A better way of utilizing development time is to look for ways to optimize using well-understood algorithms. Keep reading books (and blogs) for ways to solve common (and not-so-common) problems. A proven algorithm is well worth the time invested. Instead of inventing the wheel all over again, you can refine someone else's.

One last point: many times, the tuning of the system is left to the end of the development cycle, even after the first testing has completed. I've heard that this is done on the theory that the features should work before they get tuned. The process of keeping the system within its resources should be planned for and maintained during system development. Introducing complex changes right before you ship a product will lead to an unstable system. Balance the risk of making a premature optimization (e.g. reading all of your assembly list code looking for ways the compiler could have reduced a single instruction) against the benefit of making an optimization early so it can be thoroughly tested and have an impact on other design considerations.

Further Reading

While I have attempted to give you some tools to take care of the largest problems, there are almost always ways to squeak out a little bit more. For more optimization techniques, look to your processor vendor's application notes and your compiler manual. Atmel has a particularly good application note for one of their 8-bit processors: [AVR035: Efficient C Coding for AVR](#).

Interview question: Reverse bits in a byte

Start by reversing the bits in a byte using limited memory. Once you've finished, modify the code to be as fast as possible (but without the memory limitation).

In an interview setting, I don't mind if the interviewee starts out slow, with a relatively dumb solution. Optimization should take a backseat to correctness.

```
uint8_t SwapBitsInByte(uint8_t input) {
    uint8_t output=0;
    for (uint8_t i = 0; i < 8; i++) { // for every bit in the byte
        if (input & (1 << i)) { // need to set the bit in the output
            output |= 1 << (7-i);
        }
    }
}
```

Generally, I don't look for syntax during interviews (that is what compilers are for). However, in this problem, parentheses are important because the bit shift operator has lower precedence than addition or subtraction. If the interviewee has missed this, the reminder will probably come in the form of "walk through this test case and tell me the value of each variable at every step".

Ideally, the programmer should check her own code, running through it with a couple of values to make sure that following the instructions works. Like mental unit tests, I like to see the interviewee do this as she writes the code (or even before).

When I look at the solution, naming is important, for both function names and variable name. Comments are nice as long as they are useful ("// set the bit in the output" might be useful on an if-statement but it probably isn't on the actual setting of the bit line).

Using variables to implement the problem is a good start to the solution, although there is a straightforward way to do it with one variable (by unrolling the loop). If the candidate mentions that both variables (and input parameter) are likely to be in registers, I can point out that the use of an extra register probably meant some other variable got pushed on to the stack. Despite the correction, the interviewee gets a lot of points for the observation.

As for the second part of the problem, I don't know anyone who has successfully worked out the problem in an interview situation. It is one of those tricks that you just need to know: use a lookup table. If you can allocate 256 bytes (in RAM memory or in code space, whichever is faster), then to quickly find the reverse of a byte, all you need to do is use the value of the byte as a lookup into the array of bytes.

```
uint8_t SwapBitInByte(uint8_t input) {  
    const uint8_t lookup_table[256] = {0x00, 0x80, 0x40, 0xC0... 0x7F, 0xFF};  
    return lookup_table[input];  
}
```

This technique is often used in embedded situations when the system has more code space than time. I hope that by this point, the interviewee has mentioned that she'd look up the algorithm to make sure she had a reasonably optimal solution for the design constraints.

There is a more optimal solution to the first part of the question. Instead of reversing the bit one after another, using a temporary variable, the code can reverse the bits pairwise, then reverse the pairs, then reverse the nibbles. All without a temporary variable (and I suspect one less register can be used):

```
uint8_t SwapBitInByte(uint8_t val) {  
    val = ( val & 0x55 ) << 1 | ( val & 0xAA ) >> 1;  
    val = ( val & 0x33 ) << 2 | ( val & 0xCC ) >> 2;  
    val = ( val & 0x0F ) << 4 | ( val & 0xF0 ) >> 4;  
    return val;  
}
```

I would not expect an interview candidate to come up with this on her feet. I had to adapt it from *Hacker's Delight* by Henry S. Warren. On the other hand, now I'm tempted to make future interviewees explain what the code does (and how to make it faster, hoping they would come up with a lookup table).

When we looked at trading resources, you had to choose between RAM, code space and processing cycles. Trading resources only goes so far. Sometimes you need to know some techniques to make your code go faster. Not knowing what you'll need for your system, I can still guess that you'll need to implement some math (that being where processors excel).

The less your system does, the fewer resources it needs to do them. Sometimes we confuse important accuracy with pointless precision (see [“Accuracy vs. Precision” on page 253](#)). If you can quantify the range of data you expect and your error budget, there are some useful methods to reduce unnecessary precision for all sorts of algorithms.

Accuracy vs. Precision

Accuracy is a measure of how correct you are. *Precision* is how many digits you show in your answer. Both of them have degrees to them; one answer can be more accurate and/or more precise than another. A really accurate answer knows its limitations. For example, what is the distance to the moon?

Distance to the moon	Is it precise?	Is it accurate?
12.12345124 km	Stupidly so	No
400,000 km	No	Far more so than 12 km, accurate enough for most conversations
383,990 km	Yes	Even more so
383,990.12341231	Uselessly so	Same accuracy as the previous answer
383,990 +/- 30,292 km	Yes	Yes, best answer yet

Precision can be noise. If the extra bits of precision don't mean anything, they needlessly complicate your system, use precious RAM and waste processor cycle. As you implement complex algorithms and perform mathematics on your data, you need to deter-

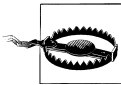
mine just how much precision is required to maintain sufficient accuracy for your product.

Identifying Fast and Slow Operations

Optimizing your system to do its mathematical operations quickly requires you to understand a bit more about your compiler and processor. Once you understand which operations occur quickly (and which ones take up one line of code but compile to use two libraries and an absurd amount of processing), you'll have the basis to optimize your system.

So, addition and subtraction are fast. Shifting bits is fast. Division is very slow. Anything with floating point is dead slow.

What about multiplication? On a DSP, it is fast: multiply and add together form a single instruction (MAC for multiply-accumulate). On a non-DSP (e.g. an ARM or your PC), multiplication is between addition and division, closer to addition.



Division isn't built into hardware as with the other arithmetic operators. It calls a library function, usually a pretty large one if you look in your map file.

Just because an operation can be described in one line of code, that doesn't mean it takes a short time to run on the processor. [Chapter 6](#) mentioned that modulo math was useful for circular buffers. It is also useful if you want to do something on every Nth pass of a loop:

```
for (i=0; i<100; i++)
    if ((i%10) == 0) {
        printf("%d percent done.", i);
    }
}
```

However, that %10 is actually a hidden division, a relatively costly instruction (though, nowhere near the processing cost of printf but that is another story). If you could make the interval a power of two, you replace the modulo math with a cheap (single instruction) bitwise logic operation (see [“Representing Signed Numbers” on page 181](#) for why).

```
for (i=0; i<100; i++)
    if (i & 7) { // this will print out every 8th pass
        printf("%d percent done.", i);
    }
}
```

In many processors, addition and subtraction take a single processor cycle. However, if they don't, then addition of unsigned variables is faster than addition or subtraction with signed variables.

Shifting bits takes a single processor cycle if the number of bits to shift is a constant. It is a cheap way to do divide and multiple, if you can keep your constants to a power of two. (We'll be discussing this one in a lot more detail.)

Using constants is faster than using variables. But they need to be real constants (`#define`) not constant variables (`const`). Now you know why embedded system programmers are still addicted to ugly `#defines`.

There are a ton of these rules but you'll learn them with time. Once you get a feel for the details of your processor (and compiler), much of it is common sense, considering what your system really needs and how to implement it. Let's take a look at a common system example to see how different choices can change an algorithm.

Taking an Average

For many signals, you'll need to calculate the average (aka mean) and standard deviation. Sometimes that is your output. Sometimes it is just a sanity check to make sure your signal hasn't gotten overly corrupted by noise.

With a rolling N-point average, you calculate the average of the last N points. You don't even need to add them all up and divide each time, you can add the new point and subtract the oldest (like a FIFO).

```
newAverage = lastAverage + (newSample/length) - (oldestSample/length);
```

However, you have to perform division at every step. And if you are averaging over a large number of small valued samples, this division may truncate and give you an inaccurate average (see sidebar on the downsides of integer math).

The other downside to a rolling average is that you need to keep the sample buffer intact until the oldest sample is leaving the average. If the average is over a large number of samples, this could be a big RAM buffer.

Instead of jumping into optimization, take a step back and consider the needs of the system. Do you need a new average value at every time step? Or can you use the same value for a period of time? If so, you might be able to implement a block average instead of a rolling average. That is where you average over a number of samples until you need the average, then you restart the calculation.

```
struct sAve {
    int32_t blockAverage;
    uint16_t numSamples;
};
int16_t AddSampleToAverage(struct sAve ave, int16_t newSample){
    ave.blockAverage += newSample;
    ave.numSamples++;
}
```

```

}
int16_t GetAverage(struct sAve ave) {
    int16_t average = ave.blockAverage/ave.numSamples;
    // get ready for the next block
    ave.blockAverage = 0; ave.numSamples = 0;
    return average
}

```

Figure 9-1 shows a set of samples with the rolling average (over five samples) and the block average (also over five samples). Note that each time the block average changes, it is the same as the rolling average result at that point. The rolling average changes faster and so it is a more accurate representation of the data at any given time step. If you only need the average at every fifth (or hundredth) time step, then the block average is just as accurate at a much lower cost in terms of RAM and processor cycles.

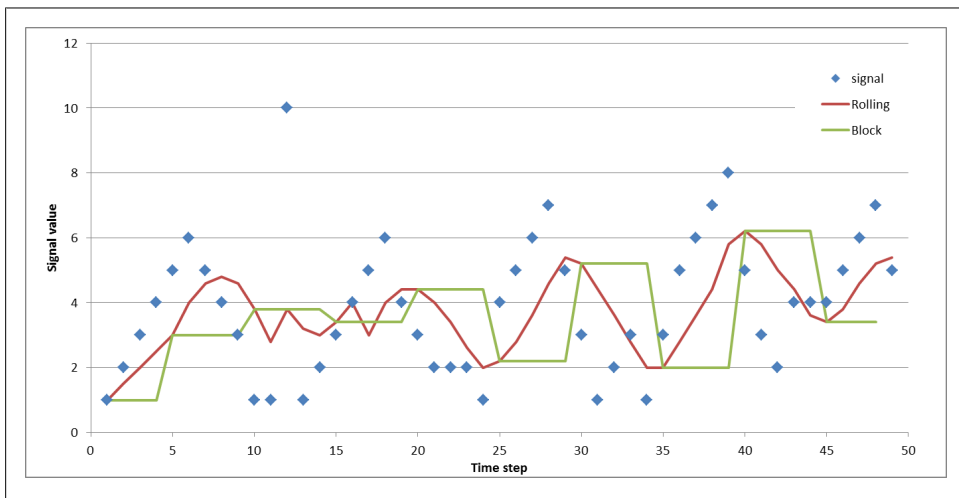


Figure 9-1. Rolling average vs block average

The Downside of Integer Math

What is 4/4? 1. Excellent. What is 5/4? Um, according to the processor, that is still 1. As is 6/4 and 7/4. Integer division truncates the number, like running the floor() function on a floating point number. It can lead to serious trouble.

For example, what if you optimized the rolling average to avoid an unnecessary division:

```
newAverage = lastAverage + ((newSample- oldestSample)/length);
```

If your samples are all about the same, your average will be horribly incorrect. Figure 9-2 shows three different forms of the rolling average. First there is the floating point version which tracks the signal reasonably well (though it is slightly delayed). Next is the sample implementation but with integer truncation as the oldest and newest sample are individually divided by the length. It tracks reasonably well but you can see some

inaccuracies. Finally, there is the implementation given above. It saves a costly division step but the result is junk.

You want to save resources but generate junk.

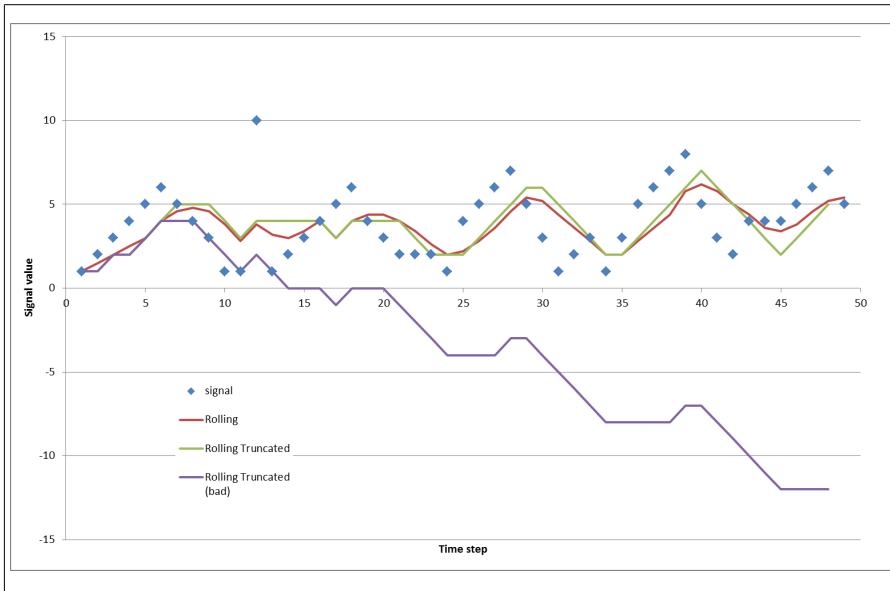


Figure 9-2. Truncation failure in rolling average

The solution is easy to state: large values should be divided by smaller ones; they shouldn't be similar in magnitude.

It is much more difficult to actually implement that. If you know that you are going to have the problem, you can choose to increase the magnitude of your numerator (that is, multiply every sample by some fixed constant). For taking an average, multiplying by any constant greater than the number of samples in the average will suffice. That would be fine here. The resulting average will be off by that amount so you may need to divide by it later.

You can choose any constant, as long as you don't multiply the samples so they are too big to fit in their variables. Not only can division cause underflow, multiplication can cause overflow which also results in odd behavior. That means you need to know the range you are expecting to receive. Using the knowledge of your system to optimize is very useful but it can lead to a brittle system where seemingly minor changes (i.e. the input range changing by 10%) lead to errors.

Thus, the downside to integer math is that you need to know and depend upon a range of input values so that you can make sure your accuracy is not degraded by the lack of precision.

Use an Existing Algorithm

Once you've got a bit of a feel for what is fast (addition, shifting, and maybe multiplication), now I'm going to share the most important thing to do when you need to implement an algorithm using minimal resources: look it up. If it is a standard algorithm, search online or pull out a numerical recipes book. If someone has already put in the time and effort to explaining how to reduce the processing cycles or the RAM usage, use their work. I mean, make sure the copyright situation is good, but most instructional materials exist so someone will use them.

There are probably several ways to implement an algorithm but only one or two will save the resources you need to preserve. For example, let's look at standard deviation.

The *standard deviation* is how far the samples in a set vary from the average of the same set. The standard deviation (σ) is calculated using each sample in the group (x_i), the mean or average of the group (μ) and the number of sample in the set (N) according to the equation in [Figure 9-3](#).

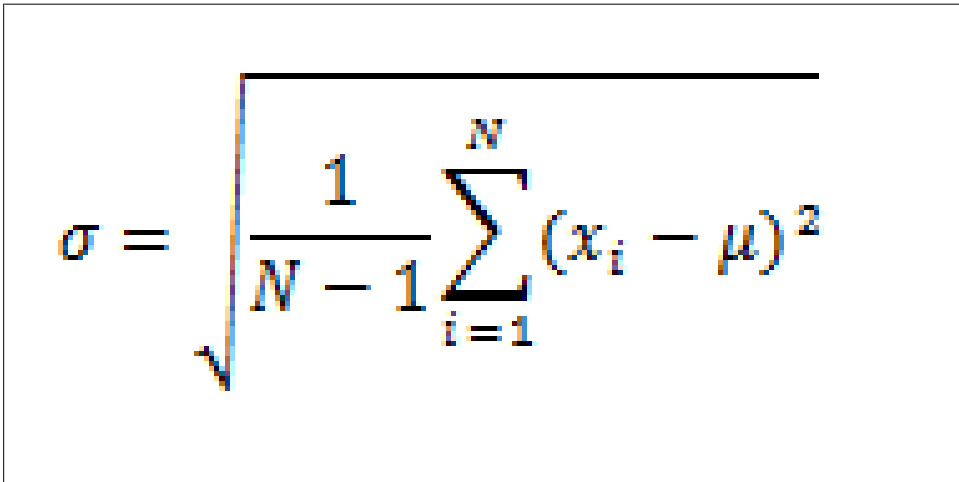

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \mu)^2}$$

Figure 9-3. Equation for Standard Deviation

How to make this friendly for your embedded system? Well, take the square root piece first... The simple answer is to not perform the square root. While it changes the value of the result, it doesn't change the information it gives you about how much the samples vary. Since many people do this, the un-square-rooted standard deviation is called a *variance*.

(Imaging how many processing cycles we've already saved by redefining the goal!)

We can start by implementing the variance pretty much as it is in the equation:

```
uint16 GetVariance(int16_t* samples, uint16_t numSamples, int16_t mean) {  
    uint32_t sumSquares = 0;
```

```

    int32_t tmp;
    uint32_t i = numSamples;

    while (i--) {
        tmp = *sumSquares-mean;
        sumSquares += tmp*tmp;
    }
    return (sumSquares/(numSamples-1));
}

```

Note that the mean was passed into the function. It has already been calculated for this block. Another piece of code has already looked at each of the samples so this function is a second pass through the data. In addition to the inefficiency of running through the loop twice, you have to keep all of the data around in a buffer, probably a waste of RAM.

However, like the block average, you can calculate variance as you go along.

```

struct sVar {
    int32_t sum;
    uint64_t sumSquares;
    uint16_t numSamples;
};

void AddSampleToVariance (struct sVar *var, int16_t newSample) {
    var->sum += newSample;
    var->sumSquares += newSample*newSample;
    var->numSamples++;
}

uint16_t GetVariance(struct sVar *var) {
    int16_t average = var->sum/var->numSamples;
    uint16_t variance = (var->sumSquares - var->sum*average)/(var->numSamples-1);
    // get ready for the next block
    var->sum = 0; var->numSamples = 0; var->sumSquares = 0;
    return variance;
}

```

There is at least one problem with that implementation. The variable `sumSquares` can get quite large (even `sum` can get pretty large if you have many samples). Instead of the signed 16-bit samples in the code, let's say the function used signed 8-bit samples. If you have two samples at 127 each, your `sumSquares` would be 32258, almost 15 bits. By the time you had five samples at the maximum value, you'd need 17 bits to hold your `sumSquares` which means moving up to a 32-bit variable. If you are using 16-bit values, the `sumSquares` needs to be a 64-bit variable. Using large variables takes RAM and processing cycles.

That wouldn't happen in the two pass implementation because the subtraction of mean from each sample keeps the sum of squares reasonably small. However, there is a way to do it without using a large intermediate variable and without taking two passes at the data:

```

struct sVar {
    int16_t mean;
    int32_t sumSquares;
}

```

```

    uint16_t numSamples;
};
void AddSampleToVariance(struct sVar *var, int16_t newSample){
    int16_t delta = newSample - var->mean;
    var->numSamples++;
    var->mean += delta/var->numSamples;
    var->M2 += delta * (newSample - var->mean); // uses the new mean
}
uint16_t GetVariance(struct sVar *var) {
    uint16_t variance = var->M2/var->numSamples;
    // get ready for the next block
    var->numSamples = 0; var->mean = 0; var->M2 = 0;
    return variance
}

```

I don't think I would have come up with that on my own, especially when I was focused on building a way to see how much the samples in my system varied from the mean. Even Knuth credits an article by another author (Welford).

Looking at the code in more detail, to store the intermediate variable for the variance, you only need a variable that is double the size of your sample (so if your sample is 8-bits, M2 can be 16-bits). This method lets you use a variable that is the same size as your data to store the mean (as opposed to the sum variable above which depended on the number of samples. However, the code is less legible and if you change the order of the lines in `AddSampleToVariance`, it will break. It also does a division operation on every pass and, worse, the division to calculate mean can cause problems because delta is likely to be small.

Yes, everything has consequences. Sometimes trying to save processor cycles is like trying to hold tightly onto a balloon; the more you grip it, the more some other area will poke out between your fingers. By using a well understood algorithm, there is a good chance that the failure analysis will already be done for you. There is a lot to be found online about optimizing some algorithms but some trusted resources described in [“Further Reading” on page 282](#) are worth having on your bookshelf when you need them.



If you really need to know the standard deviation (instead of the variance), there are many methods of approximating the square root. More than ten are listed on the relevant [Wikipedia page](#). Choose one or two then use your profiling tools to compare them with your compiler's math library.

Designing and Modifying Algorithms

Sometimes your algorithm goes beyond simple math and yet isn't something you find as a recipe in a book.



As with cooking, the more recipes you know, the more likely your modifications will succeed.

There are many tips for implementing different processor intensive operations. If you learn some of the building blocks, you can optimize for yourself.

Factor Polynomials

Say you have something that can be implemented as:

$$y = A*x + B*x + C*x;$$

That is three multiplications and two additions. If you factor your polynomial you can have it in two additions and one multiplication:

$$y = (A + B + C)*x;$$

Much better. You can take this further process further and turn the polynomial inside out:

$$A*x^3+B*x^2+Cx \implies ((Ax+B)x+C)x$$

On the left, if you'd calculated x cubed and multiplied it by A and so on you would have done nine multiplications and two additions. If you go the other way, the same answer arrives in three multiplications and two additions. How you frame a polynomial is very important.



This method is called Horner's scheme and was developed by mathematician William George Horner.

Taylor Series

Polynomials are important because they make up one way to model complex steps in your algorithm. A *Taylor series* represents a function as a sum of infinite terms. It works for most functions including trigonometric (sine, cosine, tangent, arcsine, etc), more difficult polynomials (x^{-1}), square root and logarithms. As more terms in the infinite series are added, the result becomes more precise (and more accurate). You can turn that around: you don't need infinitely many terms to generate an accurate answer for your function.

Creating a Taylor series for a generic polynomial is a fair bit of math, beyond the scope of this book (much more than the slight trickery of Horner's scheme). However, you can look them up easily. For example, the sine function can be replaced with its Taylor series expansion (see [Figure 9-4](#)).

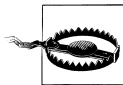
$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

$$\sin(x) \approx x - Ax^3 + Bx^5 - Cx^7, \text{ where } A = \frac{1}{3!}, B = \frac{1}{5!}, C = \frac{1}{7!}$$

$$\sin(x) = x * (1 - x^2(A + x^2(B - Cx^2)))$$

$$\sin(x) = x * (1 - x^2\left(\frac{1}{3!} + x^2\left(\frac{1}{5!} - \frac{1}{7!}x^2\right)\right))$$

Figure 9-4. Sine function in Taylor Series, rearranged via Horner's scheme



I'm going to work all of this section in radians. Remember $\pi = 180^\circ$.

For the Taylor series with four terms, the error between π and $-\pi$ is small (0.000003). If it is still too large, you can add the next term in the Taylor series for more accuracy ($+x^9/9!$). Or if you can deal with more error than that (particularly at the edges of $\pm\pi$), you can stop a term early.



Physicists often stop at the first term, approximating $\sin(x)=x$. The closer to x is to zero, the better this works.

As before, to really optimize, you need to understand your input and the acceptable error of your system. If your input is in the range of -0.2π to 0.2π , and you need an accuracy of 10%, you can use the physicists' approximation. Or if you need an error of less than 1%, you can use two terms of the Taylor expansion. A Taylor series approximation lets you balance the processing power with the accuracy required by your application (see [Figure 9-5](#)).

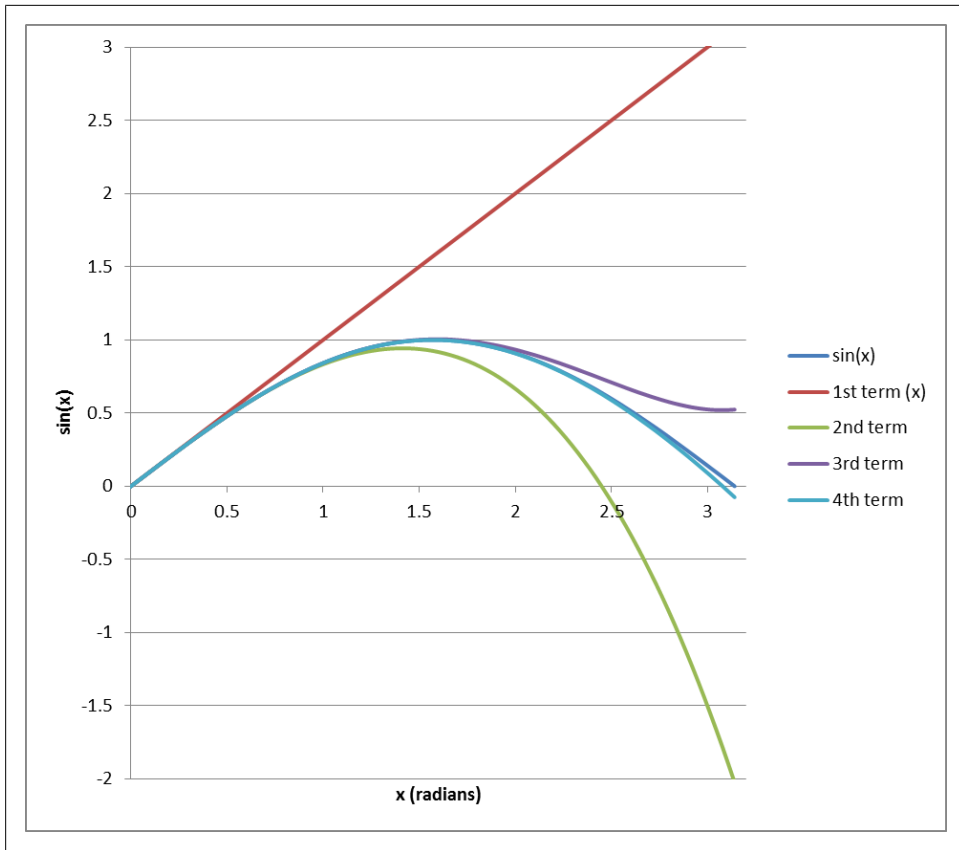


Figure 9-5. Different numbers of terms for the sine Taylor series

Because they are just multiplications and additions, Taylor series are relatively light in processor intensity. Putting in constants and using Horner's method, we can implement four terms of sine (in floating point) as:

```
xSq = x*x;
sinX = x * (1 - xSq * (INV_THREE_FACTORIAL + xSq *
                    (INV_FIVE_FACTORIAL - INV_SEVEN_FACTORIAL * xSq)));

// or taking it apart and writing multiple steps, stating with
// the end and moving toward the front:
tmp = - INV_SEVEN_FACTORIAL * xSq;
tmp = xSq * (INV_FIVE_FACTORIAL + tmp);
tmp = xSq * (-INV_THREE_FACTORIAL + tmp);
sinX = x * (1 - tmp);
```

Why does this have to be floating point? Because INV_THREE_FACT and the other constants need to be between zero and one. However, there are ways to avoid this problem.

Dividing by a Constant

Division is a relatively costly arithmetic step. But when you divide by a constant, there are ways of cheating. Using sine's Taylor series for an example, let's say you need to divide by 3!. That is $3 \times 2 \times 1 = 6$. Well, that won't be too difficult.

```
uint16_t InverseThreeFactorial(uint16_t input){
    const uint16_t denominator = 6;
    return input/denominator;
}
```

For now, let's not worry about x being between $\pm \pi$, instead taking a larger number to play with. If the input is 245, the result will be 40 in integer math (40.833 in floating point). I'm not yet concerned about the truncation here, just about the division operation. Is there a way to make it faster?

Shift is faster, much faster. Sadly, six is not a power of two. However, there are other ways of formulating the number $1/6$. I mean, $2/12$ is obvious but not useful to our purposes. What we want is multiplier/powerOfTwo to be equivalent to $1/6$ (0.166667), at least within some small amount of error. Then we can multiply by the numerator by the input and shift the result to finish the division. We trade division for a multiplication and shift. Table 9-1 shows some multiplier with their power-of-two divisors and resultant errors.

Table 9-1. Approximating 1/6 with a power of two divisor

Multiplier	Divisor	Equivalent Shift	Result	% Error
1	6	?	0.166666667	0
3	16	4	0.1875	12.5
5	23	5	0.15625	6.2
11	64	6	0.171875	3.1
21	128	7	0.164063	1.5
43	256	8	0.167969	0.78
85	512	9	0.166016	0.39
171	1024	10	0.166992	0.19
341	2048	11	0.166504	0.09
683	4096	12	0.166748	0.04

Note that the errors in the table divide in half for every additional shift. So if you implement a function to divide by 3!, it could look like this:

```
#define INVERSE_THREE_FACT_MULT 171
#define INVERSE_THREE_FACT_SHIFT 10
int16_t InvThreeFact(int16_t input){
    int32_t tmp = input* INVERSE_THREE_FACT_MULT;
    return (tmp >> INVERSE_THREE_FACT_SHIFT);
}
```

Note that the input and output are 16 bits but a 32 bit variable is necessary to hold the multiplication result. Even using a larger temporary variable, this function is cheaper than division.



If you don't do the final shift, the result can be more precise than the truncated division. You'll need to remember that the value is multiplied by 1024 when you use it.

However, there is a big downside to this implementation: you need to know the divisor ahead of time to construct the function. While it is useful anytime you need to divide by a constant, it isn't generic enough to get rid of all division.

Scaling the Input

The sine function takes an input from $-\pi$ to π and outputs a value from 0 to 1. That is all pretty difficult to do if you are avoiding floating point numbers. However, you can multiply everything by a constant to get more granularity than an integer implements. For example if you want to get rid of floating point numbers in the sine function, you can change the input to $\pm 1024 \pi$ and make the output be ± 1024 .

For mathematical operations where $f(Ax) = Af(x)$, it is as simple as using the scaled input. Even for functions that are $f(Ax) = f(A)f(x)$, removing the scalar is straightforward (divide by $f(A)$). However, sine is more complex than that.

You'll have to keep track of the multiplications in the Taylor expansion which means checking that each step is only scaled by a single 1024 and not multiple:

```
xSq = x*x >> 10; // right shift is like divide
tmp = - InverseSevenFactorial(xSq);
tmp = (xSq * (INV_FIVE_FACTORIAL + tmp)) >> 10;
tmp = (xSq * (-INV_THREE_FACTORIAL + tmp)) >> 10;
sinX = x - ((x * tmp)>>10);
```

The great news here is that with the scaling, the division at some of the steps becomes easier. Everything must be multiplied, including the constants. Instead of multiplying by $1/3!$ (essentially dividing by 6), we are multiplying by $1024/3!$ which is about 171. Since $5!$ is 120, there is more error to rounding $1024/120$ to 8 (8.533). The last term ($7! = 5040$) is still large so it requires some actual division, though it is still dividing by a constant. Because it is a small constant, there is some error associated with it as shown in [Figure 9-6](#).

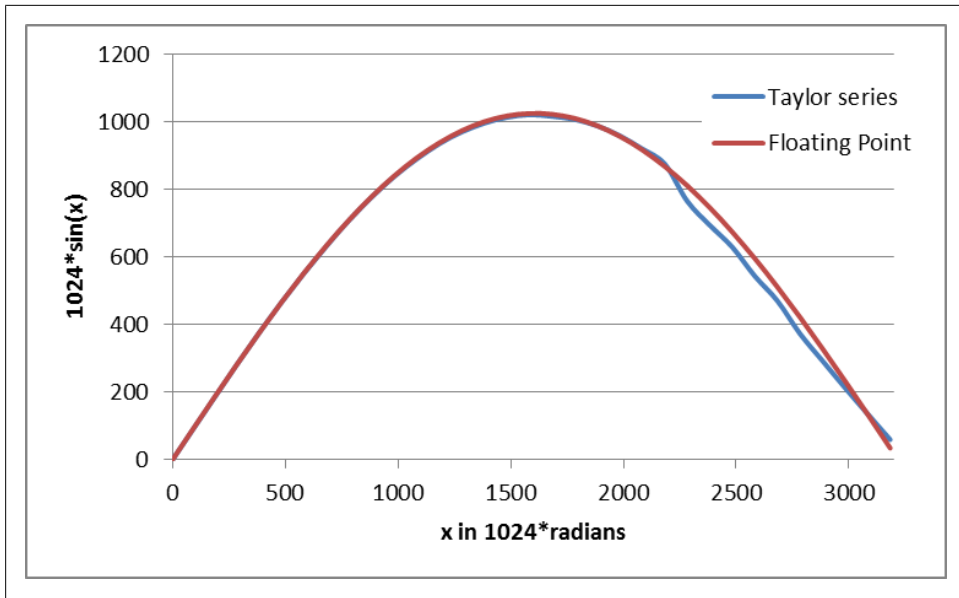


Figure 9-6. Errors in scaling the input with Taylor series (0 to π)

There are ways to alleviate these errors (such as scaling by a larger number). As you are implementing your changes, you will need to understand where error comes into the system. As with profiling, the bulk of your time should be spent on the sources of the largest error, not optimizing a minor contributor.

These ideas of scaling and division by shifting are going to help us create a method to avoid floating point numbers entirely. Before we go directly there, let's look at a very popular way to save processor cycles.

Lookup Tables

As mentioned in [Chapter 8](#) (and expounded on in the interview question), lookup tables are blazingly fast and just require a bit of code space.

However, to use a lookup table, you need to know the range of the input and the acceptable error. In Taylor expansions, it drives the number of terms you calculate. In lookup tables, these parameters determine the number of entries in your table.

Implicit Input

Each entry in the look up table requires two pieces of information: the input (x) and the output (y). However, not all look up tables require both to be in the table. With implicit input lookup tables, the position in the array indicates the input value. For example, here is a sine lookup table with an input in milliradians ($-\pi/1000$ to $+\pi/1000$)

and an output also scaled by 1000. (Note: lookup tables decrease the dependency on power of two shifts, often leading to more human-sensible scaling.)

```
const int16_t sinLookup[] =
    -58,  // x = -3200
    -335, // x = -2800
    -676, // x = -2400
    -910, // x = -2000
    -1000, // x = -1600
    -933, // x = -1200
    -718, // x = -800
    -390, // x = -400
    0,    // x = 0
    389,  // x = 400
    717,  // x = 800
    932,  // x = 1200
    999,  // x = 1600
    909,  // x = 2000
    675,  // x = 2400
    334,  // x = 2800
    -59   // x = 3200
};
```

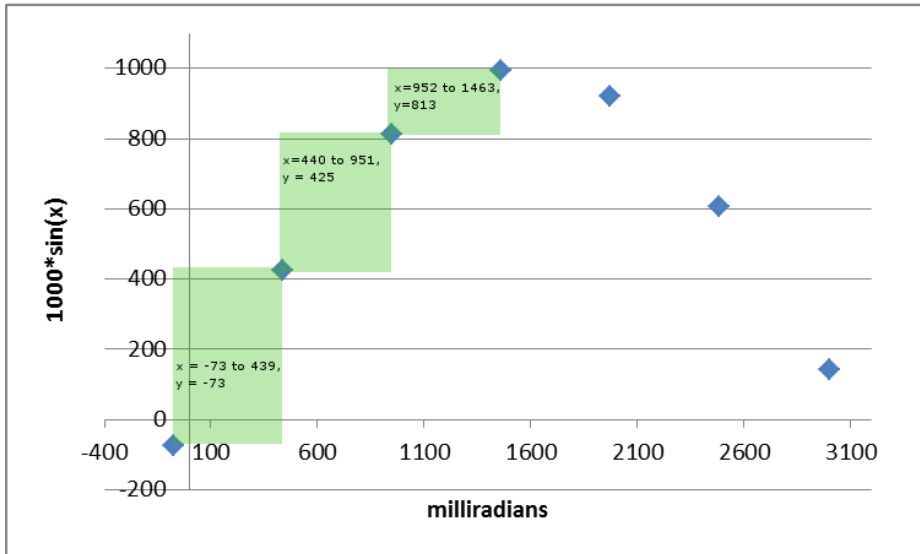
To use the table, you'll need to calculate the index that matches your input. To do that, subtract the lowest value in the table and divide by the step size of the table:

```
uint8_t index = (x - (-3200))/400;
y = sinLookup[index];
```

Wait a minute, why are killing ourselves optimizing a sine function if we are just going to do a division now? Right. The step size of the table not only changes the precision of your table, it can also change your processing requirements. The non-power of two step size was a bad choice. If we'd chosen 512, the table would only be a little bit smaller (less precise) but the processing would have been shorter:

```
uint8_t index = (x - (-3145))>>9; // use #defs, not magic numbers
y = sinLookup[index];
```

Given only thirteen entries, the table is not very precise but will the results be accurate? [Figure 9-7](#) shows points and where they'll land in this calculation. With this code, we are looking at a) where the value returned by the table represents the result of the input at the start of the interval covered. The table would be more accurate if the output value covered the center of the range as shown in b).



- a) Incorrect left biased lookup points (above)
b) Corrected center biased lookup points (below)

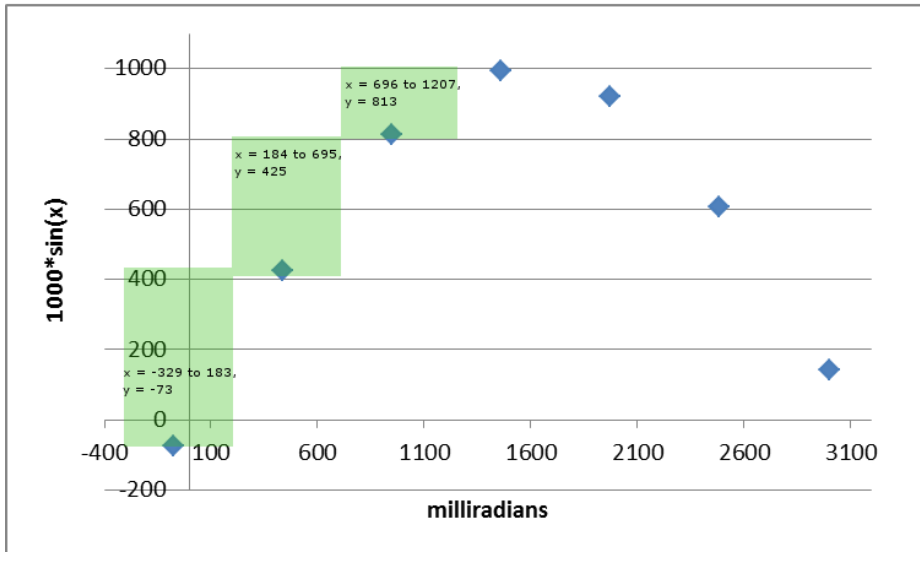


Figure 9-7. Comparing Sine

The table itself doesn't need to change to make this happen, just the indexing. In effect, we need to add half a step when calculating the index:


```
uint8_t index = (x - (-3145 - 256))>>9;
y = sinLookup[index];
```



This idea of adding a half step to center the result is prevalent in almost every well implemented lookup table.

Once you change the valid range, I'd recommending making the comments describe the range as well as the value.

```
#define BASE_SIN_LOOKUP (-3124 - 256)
#define SHIFT_SIN_LOOKUP 9
const int16_t sinLookup[] =
    3,      // x @ -3145, for range -3401 to -2888
    -487,   // x @ -2633, for range -2889 to -2376
    -853,   // x @ -2121, for range -2377 to -1864
    -1000,  // x @ -1609, for range -1865 to -1352
    -890,   // x @ -1097, for range -1353 to -840
    -553,   // x @ -585,  for range -841 to -328
    -73,    // x @ -73,   for range -329 to 182
    425,    // x @ 493,   for range 183 to 695
    813,    // x @ 951,   for range 695 to 1207
    994,    // x @ 1463,  for range 1207 to 1719
    919,    // x @ 1975,  for range 1719 to 2231
    608,    // x @ 2487,  for range 2231 to 2743
    142,    // x @ 2999,  for range 2743 to 3255
};
```

Note that this new table has a step size of 512 and thirteen entries. It looks [Figure 9-8](#). This table covers the expected input ($-\pi$ to $+\pi$) and a little more besides. However, if there is a possibility of getting data outside the table's range, add code to verify the inputs. After all if your input is 2π , then your index will be 19, outside the range of the table. However, accessing `sinLookup[19]` will work in C but give you a completely incorrect result.

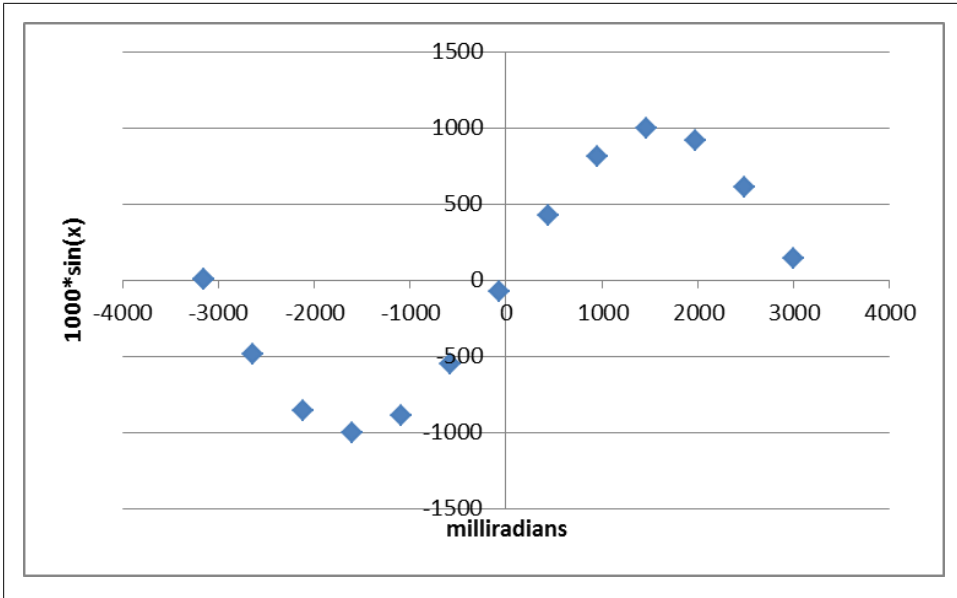


Figure 9-8. Points in sine lookup table

Linear Interpolation

In the lookup table, even with the center biasing, there are times when your result should be more accurate than a single number covering a range of inputs. In that case, linear interpolation may be the way to go.



You can use other interpolation methods. Consider the trade-offs between the increased computational requirements of a polynomial fit against the code space of putting in a larger table.

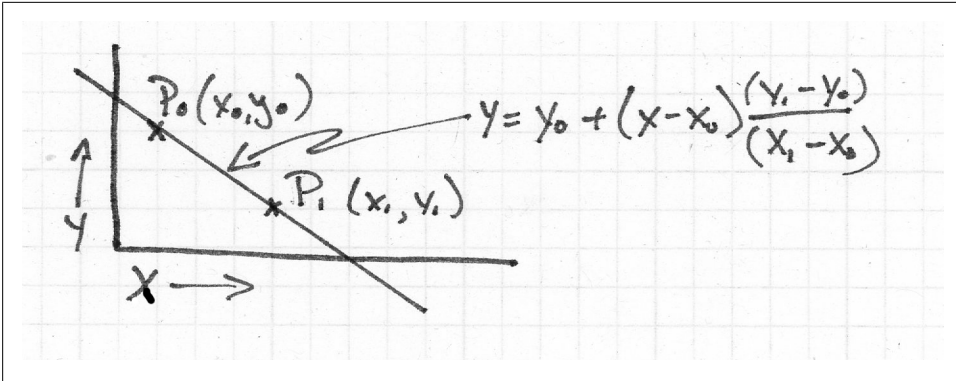


Figure 9-9. Linear Interpolation

As shown in [Figure 9-9](#), linear interpolation isn't hard, just a bit of algebra you might not have used for a while. You'll need two values from the lookup table: the one below the input you have and one above. In the figure, the goal is to find the output for $x=5$ so the two closest points are used. Since the input and output values are linked, for interpolation, I find it simpler to treat them as points instead of individual values:

```
struct sPoint {
    int16_t x;
    int16_t y
};
```

The linear interpolation code is simply math, be sure to keep the bits from overflowing with the multiplication.

```
// This linear interpolation code does:
// y = p0.y + ((x-p0.x)*(p1.y-p0.y))/(p1.x-p0.x);
// But in a bit safe way.
int16_t interpolate(struct sPoint p0, struct sPoint p1, int16_t x)
{
    int16_t y;
    int32_t tmp; // start and end with int16s but can need a larger intermediate
    tmp = (x - p0.x);
    tmp *= (p1.y - p0.y);
    tmp /= (p1.x - p0.x);
    y = p0.y + tmp; // now safe to go back to 16 bits
    return (y);
}
```

This method works even if the input is not between the two points. If you end up with an input that is just outside your table, you can use the last two points in the table to interpolate past the end.

Note there is a division associated with linear interpolation. You are trading some processor cycles to get more accuracy with a smaller table size (code space). You can avoid division by making size of the x-steps in the lookup table a power of two (as I did with

the last sine lookup table above). Then the divisor is a power of two so the divide can become a shift.

While the table lookup depends on the table itself, interpolation is a step away from that. By optimizing the division out, your interpolation function depends on the table data. If you have multiple tables, you'll need multiple interpolate functions (or a parameter to describe the shift value). Be careful as it will make your code less readable and more brittle, linking data and code together. If you need the processing cycles, it may be the best way. But it makes for lousy code.

Explicit Input in the Table

For some functions, you can create a smaller or more accurate table by allowing variable step sizes, interpolating between them. The sine function is very linear toward the center as seen in [Figure 9-8](#). We could easily take out the three center most points and let linear interpolation handle it (which is why $\sin(x) = x$ for small x is so popular). Even the outer most edges could have a sample removed; it is those tricky curves that need more points.

Sometimes a lookup table has a variable step size to decrease errors in certain areas. Linear interpolation can be used between the areas as shown in [Figure 9-10](#).

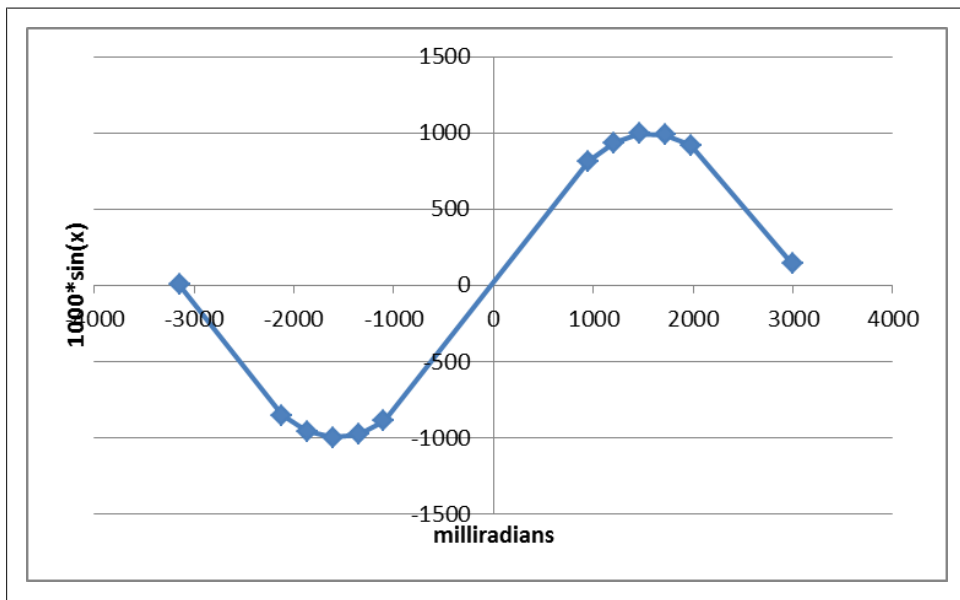


Figure 9-10. Using linear interpolation between points on an explicit lookup table

The lookup table is now nearly double in size because we have to put in the start of the range covered as well as the value. It takes up more code space but we gain accuracy in the areas we need it most

```

struct sPoint sinLookup[] ={
    { -3145 , 3 },
    { -2121 , -853 },
    { -1865 , -958 },
    { -1609 , -1000 },
    { -1353 , -977 },
    { -1097 , -890 },
    { 951 , 813 },
    { 1207 , 934 },
    { 1463 , 994 },
    { 1719 , 989 },
    { 1975 , 919 },
    { 2999 , 142 }
};

```



An explicit input lookup table is often used if you have to build the table from the environment such as when calibrating a sensor for temperature effects. The x value would be the temperature read via a thermistor and the y value would be the offset from the expected value.

Unfortunately, using an explicit table means you need to search through the entries to find the correct input range. As long as the table is in order, the search is straightforward. The goal is to find the index into the table with the closest x value without going over.

```

int SeachLookupTable(int32_t target, point_t const *table, int tableSize)
{
    int i;
    int bestIndex = 0;

    for (i=0; i<tableSize; i++) {
        if (target > table[i].x) {
            bestIndex = i;
        } else {
            return bestIndex;
        }
    }
    return bestIndex;
}

```

You can pair this with interpolation:

```

index = SeachLookupTable (x, sinLookup, sizeof(sinLookup));
if (index+1 < sizeof(sinLookup)) {
    y = interpolate(x, sinLookup[index], sinLookup [index + 1]);
} else {
    y = interpolate(x, sinLookup[index-1], sinLookup [index]);
}

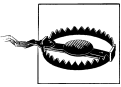
```

Unlike with the implicit input, this method allows the input to be beyond the range of the table. It will linear interpolate using the first two points (if the input is less than the

first entry in the table) or the last two points (if the input is greater than the last look-up point).

Fake Floating Point Numbers

I've been showing my biases and taking for granted that you won't be using floating point numbers. If you have a floating point unit (FPU) in your processor, then that is a poor assumption on my part. If you don't, then now might be a good time for you to look in your map file to see how much code space the floating point libraries take up just to, say, add two numbers together. For a processor optimized for space, modifying an integer add to use floating point took 532 bytes of code space (to add the library functions `__aeabi_fadd` and `__aeabi_fsub`, even though I only used the add function).



Standard input and output functions often include floating point handling (i.e. `printf` or `iostream`). Even if you don't use floating point numbers in your code, if you use these library functions, your code will still include the floating point math libraries. Use your map file!

Of course, code space is only part of the problem with floating point numbers: compared to integer math, floating point math is slow. Floating point numbers are expensive in an embedded system. Unless you've got a really good reason to do otherwise, avoid them like the plague. If you can't avoid them, you can fake them.

To get the idea, remember back to grade school, when there weren't floating point numbers in your life. Even before we could write 0.25, we knew what a quarter of a pie was and that it could be written as one over four ($1/4$). Any rational number can be written as a fractional number (ratio of a numerator over a denominator). Even irrational number can be approximated with less error coming from having a larger denominator. (For example, π is often approximated with $22/7$.)

If we stopped there, it wouldn't be such a big improvement (divides are slow). Except, we've already gotten around divides by shifting by a power of two. I didn't give it a name before but this technique is called *binary scaling* and forms the basis for our fake floating point numbers. We'll start with a 32-bit numerator and an 8-bit exponent. The exponent is the number of bits to shift to the left (for positive values) or to the right (for negative values).

We'll start off by defining a structure to hold the values:

```
struct sFakeFloat {
    int32_t num; // numerator
    int8_t shift; // right shift values (use negative for left shift)
}
```

The number held in this structure is represented by

```
floatingPointValue = num / 2shift;
```

And now for a few examples. Let's start by going back to a quarter of a pie. The numerator is one, and the denominator is already a power of two.

```
struct sFakeFloat oneFourth = {1, 2};
```

A negative shift value changes the direction of the shift. We could rewrite four using this notation as well:

```
struct sFakeFloat four = {1, -2};
floatingPointValue = four.num >> shift
==> 1/(2(-2)) ==> 1 * 22 ==> 4
```

Fake floating point can also be used to work with numbers larger than normally stored in a 32-bit value (or whatever the size of the numerator). The result will be less precise than a larger variable would be. For example, say we want to store ten billion and one (10,000,000,001). We can make the value fit into a signed 32-bit integer by dividing by eight:

```
struct sFakeFloat notTenBillionOne = { 1250000000, -3 };
```

However, while this is within one ten billionth of being correct, we've lost the last digit; the resulting value is only ten billion.

Precision

In the structure, we could use a 24-bit numerator and an 8-bit shift for the denominator. While it would keep everything in a convenient 32-bit variable, it might decrease precision. We've already seen how we can lose digits for big numbers but the same can happen with other numbers.

For example, the number 12.345 could be represented as 49/4 with an error of 0.095. With a larger denominator shift value, more precision can be obtained. However, a too large denominator will cause the numerator to overflow.

Table 9-2. Representing the number 12.345 using binary scaling

Numerator	Number of bits needed in numerator	Denominator shift values	Equivalent floating point number	Error
12	4	0	12	0.345
25	5	1	12.5	0.155
99	7	3	12.375	0.030
395	9	5	12.34375	0.00125
12641	14	10	12.34472656	0.000273
12944671	24	20	12.34500027	2.67E-07
414229463	29	25	12.345	1.19E-09
1656917852	31	27	12.345	1.19E-09

So as you increase the shift in the denominator, you can decrease the error but you'll also need to increase the number of bits in the numerator. The question returns, how much error can your system tolerate for this mathematical operation?

Also, as before, you need to be concerned with overflowing the bits (mostly with multiplication though it can happen with addition).

Addition (and Subtraction)

Let's look at addition for a pair of fake floating point variables. Again, thinking back to grade school days and fractional math, you need to make their denominators the same before trying to combine numbers. In this case, it means making the shift values the same, generally by left shifting the smaller number before doing the addition.

The best way to explain is to walk through an example. In this example, we'll use a byte for floating point numerator. This can be useful when not a lot of range is needed. Looking back at our 12.345 table, using 8 bits we got an error of 0.030 or less than 1%. (Also, using an 8-bit numerator means you can imagine overflows without huge numbers.)

```
struct sFakeFloat {
    int8_t num;
    int8_t shift;
};
```

This example is only going to show addition using positive numbers but the method is mostly the same for subtraction and for negative numbers. First we need to set up our floating point numbers with some values.

```
struct sFakeFloat a = { 99, 3 }; // 12.375, not quite 12.345
struct sFakeFloat b = {111, 5 }; // 3.46875, not quite 3.456
struct sFakeFloat result;        // 15.84375, close to ideal of 15.831
```

Next, find the least common denominator so we can add them together. Finding the least common denominator when all denominators are powers of two is very easy. Determine which is larger and then elevate the other value.

```
int16_t tmp = a.num;
tmp = tmp << (b.shift - a.shift);
```

A larger temporary variable must be used or the result may overflow and get truncated. This code should check that the difference in powers of the denominator (`b.shift - a.shift`) is less than eight bits (or less than 24 bits if the temporary is a 32-bit variable). In addition to shifting the variable with the smaller denominator up, you may also need to shift the variable with the large denominator down (dividing by two and losing precision).

The sum of the numerators is kept in the temp variable until we make sure it is small enough to fit into our result variable.

```
tmp = tmp + b.num;
```


At this point the shift value is the larger of the two (`b.shift = 5`) and the temp variable is 507, too large for an 8-bit variable. Until this temporary variable can safely fit back into the variable size we have, we'll need to make the numerator smaller (by dividing by two).

```
result.shift = b.shift;
while (tmp > INT8_MAX || tmp < -INT8_MAX) {
    tmp = tmp >> 1;
    result.shift--;
}
result.num = b.num;
```

Finally, we have a result that describes the addition of two floating point numbers.

The result's numerator is 126 and the shift is 3 for a floating point equivalent of 15.84 (plus some unnecessary precision). This is no different than if we'd added the two numbers in floating point (other than the initial precision error in representing).

Multiplication (and Division)

Multiplication is a little simpler because you don't have to make the denominators match. In fractional math, you just multiply the top and the bottom separately to get the answer ($2/3 * 5/3 = 10/9$).

When you multiply two fake floating point numbers, multiply the numerators safely, then multiply the denominators (sort of).

We again start by putting the numerator of one in a larger temporary variable so we don't get an overflow. Unlike addition where we needed to check the size, we can be certain that the result of two 32-bit variables will be no more than 64 bits.

Then multiply the temporary variable by the numerator of the second.

The shift value is not exactly the denominator; it is the exponent of the denominator:

```
denominator = 2^shift;
```

Instead of multiplying the shift values, we only need to add them together. If the resulting numerator is greater than the number of bits we have available, shift to the right until it fits.

For this multiplication example, we'll return to `int32_t` to hold the numerator.

```
struct sFakeFloat {
    int32_t num;
    int8_t shift;
};
struct sFakeFloat a = { 1656917852, 27 }; // 12.345
struct sFakeFloat b = {128, 8}; // 0.5
struct sFakeFloat result;
```

For this example, we'll give the variables precise values so we can check the process at each stage. (Though one half could be trivially implemented at $\{1, 1\}$).

The temporary variable noted must be at least as many bits as both numerators combined. Since we are using 32-bit numerators, the temporary variable has to be 64 bits. If we were using 16-bits in the numerator, we'd need at least 32 bits to hold the multiplication. Note that a 64 bit integer may not be native to your processor so using it will incur some overhead but not nearly as much as there would be if you did a variable of type float or double.

```
int64_t tmp;  
tmp = a.num;  
tmp = tmp * b.num;
```

The multiplication is done in two steps to upgrade the variables into the larger sized variables. If you'd just multiplied the numerators together, the result would have been an `int32_t` before it got upgraded and stored in an `int64_t`. You need to upgrade, then multiply.

```
result.shift = 0;  
while (tmp > INT32_MAX || tmp < -INT32_MAX) {  
    tmp = tmp >> 1;  
    result.shift--;  
}  
results.num = tmp;
```

As with addition, as long as the results of the numerator multiplication don't fit in the results numerator, we need to increase the shift to avoid overflow. For the values above, it decreases the shift eight times.

We can also make sure that the shift values don't overflow the shift variable:

```
tmp = a.shift + b.shift;  
ASSERT(tmp < INT8_MAX);  
result.num = tmp;
```

The result of the operator is 6.1725 {1656917852, 28}.

Division is a bit more complicated but works along the same lines as multiplication where the numerators are divided and the shifts are subtracted instead of added.

Note that both addition and multiplication have places where they can have errors. You'll need to limit your variables or return those errors to a calling function.

Determining the Error

One of the difficulties with using binary scaling is recognizing the limitations. The results will be the best if you have plenty of prior knowledge about the numbers you'll be working with so you can handle overflows and verify the stability of your system. You could make a very generic library to handle every possible case but you run the risk of re-implementing floating point numbers. As you look at your algorithm, you'll need to know the range (min and max) of the variables and the precision you need for dealing with them at each point in time.

Say you have a system where you need to perform a function on the input from an ADC:

$$y = Ax^2 + Bx + C$$

where x is the ADC value.

Making up some other realistic details for this example, the ADC is 10-bits so the minimum value is 0 and the maximum value is 1024 and we want the output to have less than 0.01% error. The coefficients A, B and C are set in the factory as a calibration. We have control over the manufacturing process and can fail a unit in the factory if the coefficients don't fall within the tolerated range. See the table for the other information.

Table 9-3. ADC example limits of parameters

	Input from ADC	A	B	C
Minimum	0	-0.001	0.002	-2
Maximum	1024	0.001	0.2	3

We can determine the maximum output value (in floating point) using the maximum coefficients in the equation. The minimum is a little more complicated because A can be negative such that the maximum ADC value used with the minimum coefficient is less than the minimum ADC value and the minimum coefficients.

First, we make sure we can hold our range of outputs in a binary scaled floating point value by trying different shift values and calculating the error. Table 9-4 shows an example where the scaling is -1, indicating the numbers will be shifted to the left by one place (so everything is multiplied by 2).

Table 9-4. Example of error examination (bad result)

	Minimum result	Maximum result	Notes (and Excel formulas)
$A*ADC^2 + B*ADC + C$	-1048.528	1256.376	Calculated from the parameters in Table 9-3.
Shift value	-1	-1	Try different ones to get to an error level that is acceptable.
Numerator	-2097	2513	$=ROUND(result*2^{shift})$
Bits needed in the numerator	$12+1 = 20$	$12+1 = 20$	$=CEILING(LOG(numerator, 2)) + 1$ (if signed)
Variable size of numerator	16	16	Needs to be greater than the previous line. Note that signed values require one more bit.
Precision granularity	0.5	0.5	$=2^{shift}$
Absolute error in calculation	0.028	0.124	$=ABS(result-(numerator / 2^{shift}))$

The error is about 3% with the minimum result and about 12% using the maximum, pretty far off from our goal of 0.01%. How do we go forward? It requires some judgment but here is the basic recipe.

First, determine the range of the variable. Note whether the inputs and outputs are signed or unsigned.

For each input and output, check the representation and verify the granularity is acceptable. To do that, start by determining how many bits are in your numerator. Ideally, this is the number of bits native to your processor: 8, 16, or 32. You can make it smaller so that your whole structure fits into 32 bits but you run the risk of creating more (and slower) code as the assembly level tries to shift the variable around. Next, choose a shift value that allows your numerator to fit in the number of bits you have available. Note if there is a range of acceptable shift values. Verify the number can be represented in a binary scaled format at the extreme values without overflowing. Verify the granularity is acceptable; it should be less than your goal error. Finally, look at the errors at the extreme value and for a nominal value. These will always be less than the granularity but it provides another check that this scaling value is valid.

Considering this, we can get much better shift values as shown in [Table 9-5](#). It is an iterative process so I tend to use Excel or some other spreadsheet program to help calculate the values.

Table 9-5. Example of error examination (good result)

	Minimum result	Maximum result	Notes (and Excel formulas)
$A*ADC^2 + B*ADC + C$	-1048.528	1256.376	Calculated from the parameters in Table 9-3 .
Shift value	-8	-8	Try different ones to get to an error level that is acceptable.
Numerator	-268423	321632	$=ROUND(result*2^{shift})$
Bits needed in the numerator	$19+1 = 20$	$19+1 = 20$	$=CEILING(LOG(numerator, 2)) + 1$ (if signed)
Variable size of numerator	31	31	Needs to be greater than the previous line. Note that signed values require one more bit.
Precision granularity	0.00390625	0.00390625	$=2^{shift}$
Absolute error in calculation	0.00065625	0.001	$=ABS(result-(numerator / 2^{shift}))$

We can represent the range of interest with very little error inherent in the fixed point number. While not required, it is good that we can represent both the minimum result and the maximum result with the same shift value, it means we can choose to optimize later. So that covers the minimum and maximum of the output, now we need to go through the same process for each input and determined how big the variables need to be between each step.

Since the A coefficient gets multiplied by the square of the ADC input ($1024*1024$), any error in its representation is going to get multiplied by a million. Let's look at what the range of the coefficient might be and how that might change the representation needed as shown in [Table 9-6](#).

Table 9-6. Example of error examination on the coefficients

	Minimum	Maximum	Notes and Formulas
Coefficient A in floating point	-0.001	0.001	In manufacturing, we'll reject any calculated coefficient that does not fit in this range.
A shift	32	32	Choose this so that the numeric error is small.
A numerator	-4294967	4294967	$=\text{ROUND}(\text{coeff} * 2^{\text{shift}})$
Actual bits in numerator	$23 + 1 = 24$	$23 + 1 = 24$	$=\text{CEILING}(\text{LOG}(\text{numerator}, 2)) + 1$ (if signed)
Granularity	4.2950E+09	4.2950E+09	$= 1 / (2^{\text{shift}})$
Error associated with coefficient A	6.89179E-11	6.89179E-11	$= \text{ABS}(\text{coeff} - (\text{numerator} / 2^{\text{shift}}))$
Multiplied by max input value (1024*1024)	-4.5036E+12	4.5036E+12	Multiply out the input and the coefficient.
Number of bits required to hold the results	44	44	Note that this will not fit in a 32-bit integer unless we shift it down.
Floating point $A * x^2$	-1048.576	1048.576	Using floating point, this is the answer we expect.
Faked point $A * x^2$	-1048.575928	1048.575928	Using fixed point this is the answer we get.
Error between floating and faked	7.22656E-05	7.22656E-05	Error due to using fixed point. This can be made smaller by increasing A shift.

Note that the A part requires 44 bits (43 + a sign bit) so during calculation, the intermediate value will need to be an `int64_t`. At the end of calculating Ax^2 , we can keep the intermediate variable around for further use or put the A part back into our structure. In this example, to go back in the structure, it needs to fit into a 32 bit signed integer so it will need to be right shifted by (44-32=) 12. Fortunately, this makes no difference for the error. If it did, we might keep the intermediate value around until the shift was safe. (If it was never safe, our original check would have shown us that the result could not be represented in this way.)

So once we do the same for B and C (both a little easier), we can add up the errors for each part to the maximum error expected. If it is outside your error budget, look at the largest error sources and increase their shift values (until they don't fit into the variable size, then look at the next largest error source and increase their shift value and so on).

The next step is to add each part of the equation. In this example, it is straightforward but you do have to remember to shift them all to the same shift value so they have the same denominator before adding the numerators.

As part of adding these together, we have to check that the variable we are using to store each operation is large enough. If this part of the equation used all 64 bits of our variable, then we'd have to shift down before doing the next operation and we'd have to check that the shift down did not increase the error outside the acceptable range. The addition will require an intermediate because adding two 32 bit integers may lead

to a 33 bit integer. After the addition, check that the number can be shifted to fit back in the structure.

The order of operations and shifting is important but very application specific. You must determine where the overflows (and underflows) occur or you'll end up with some fairly strange behavior.

The last part of the process is to check the final fixed point formulated numbers against the expected floating point version. The error here may be different (smaller) from the added together error above because the errors don't always combine in a maximally bad way (they tend to cancel out).

This process of determining the possible error is laborious and often requires multiple passes as other parts of the system change. Creating a well-documented spreadsheet will save you time.

Further Reading

Someday I'd like to sit down and read the Art of Computer Programming. The scholastically inclined part of me is slightly stunned I haven't done it already. The more pragmatic (um, lazy) part convinces me to just look up what I need, when I need it and not get too distracted by all of the interesting, shiny object around in the book. Whether you've read it or not, it should be on your shelf: Knuth, Donald E. (1998). The Art of Computer Programming, volume 2: Seminumerical Algorithms. Addison-Wesley. ISBN 0-201-89684-8.

Alternatively, you may want to use a numerical recipes book for more implementation specific information: Press, et. al (2007). Numerical Recipes 3rd Ed: The Art of Scientific Computing. Cambridge University Press. ISBN 0-521-88068-8. Variations come in language specific forms (C, C++, Java, etc).

At the deeply embedded scope there is the Warren, Henry (2002). Hacker's Delight. Addison-Wesley Professional. ISBN 0-201-91465-8. This is an interesting book to flip through, almost a grimoire of slightly scary optimization techniques. Compiler designers tend to need to know these things. You want people to be able to read and reuse your code so don't go too overboard in optimization at this level.

Interview question: Handling Very Large Numbers

Write a program to add the numbers from 1 to 10 in the language of your choice, use C if you don't care. As the interviewee complete each piece, a new question is asked: What do you need to do to make it generic so it is 1 to n? What sorts of optimizations can you provide? What limitations are inherent in the implementation? How would you change it for the input to be arbitrary length? (Thanks to Rob Calhoun for this one!)

As the interviewee starts with the first question, does she create a function that takes a parameter to stop the addition (1 to n)? That is better than hard coding it immediately

to go from 1 to 10. From there, I look at the size of variables for both input and output variables- using integer or long variables is ok, but can she explain the limitations it places on the code? Does she use unsigned or signed variables and can she explain why unsigned are preferable?

If she doesn't know the formula for the sum from 1 to $n = n*(n+1)/2$, I help her through it typically by writing 1, 2, 3, 4, 5, 6 out and pairing up 1+6, 2+5, 3+4. The interviewee loses no points lost for not knowing this formula as long as she's awake and figures it out. Points are lost for interrupting and insisting upon an incorrect formula, even in the face of counter examples (it happens more often than you'd think).

Next, we go back to the size of the variables. What is size of $n * (n+1)/2$ compared with n ? If n is a 16 bit integer, how big does the output need to be? What if n is 64 bits? I prod to ensure she knows that $(2^{16})^2$ is $2^{(16*2)}$.

Finally, we get to the really interesting part of the question where I get to ask how she'd change it if it needed to work for inputs up to 2^{64} . Now what can she use as an output? If she picks double-precision floats, well, we discuss the limitations of floats. There is no wrong answer here, it is building up- what if we need to use even larger numbers? I limit it to on the order of a few hundred...or a few thousand...digits, always constraining it to not be more than a million hexadecimal digits.

Now how can the interviewee modify her function? I walk them through constructing the data structure used for the input buffer and the output buffer (and she should know by this point how big to make the output buffer: twice the input) and the function prototype.

I look for thread safety, failure to keep track of the size of the byte string, discussing as necessary. Finally, I get them get her to do as much of the implementation as she can of computing $(n * n - 1)/2$ for arbitrary length integer, reading from `*input` and writing to `*output`. If she's are having some trouble, I suggest using an 8-bit machine as it is more straightforward to draw out.

Depending on the skill the interviewee shows, this question can move in different directions, drilling down to show strength and weakness. Usually, it takes the whole interview slot to work through only part of it.

Reducing Power Consumption

Whether you are building a device that fits in your pocket or trying to save the world by reducing your company's carbon footprint, decreasing a system's power consumption can take an order of magnitude more time than implementing the product features. Choosing all the right hardware components is a huge part of making a system power efficient. But because the processor is likely to be one of the largest consumers of power in the system, software can play a big role in saving electricity.

The pressures to decrease power usage and cost are what lead us to select processors that don't have enough resources to comfortably perform the product features. Since you are a relatively expensive resource, using your time to save on cost of the processor is sensible only if you are building enough units to amortize your time.



How expensive are you? Take your annual salary and divide by a thousand. That is about what each hour of your time costs your company, counting salary, benefits, office space and all the little things that add up. So if you are making \$80,000 per year, each hour is worth about \$80. If you can buy a \$250 tool to tool to save four hours, it is usually worth it. Of course, there is a difference between capital outlay (cash) and sunk cost (your salary) so while this is a good rule of thumb, your boss may not let you buy the scooter to get from your desk to the break room even after you describe the cost benefit of saving thirty seconds every day.

On the other hand, a processor with the minimum amount of power cycles, RAM, and code space consumes less power than one with plenty of resources. As suggested in [Chapter 8](#), you may want to optimize to "as low as possible," but that will take an infinitely long time. Start with a quantifiable system goal. Let your electrical engineering team do their part when choosing the components. Then use the points in this chapter to reduce the power further.

Understanding Power Consumption

All electrical circuits can be modeled as resistors, though this is akin to saying all differential equations can be modeled linearly. It is true, but depending on the circumstances, you might expect to see a lot of error. However, it is a good start. Power is measured in watts and is proportional to the square of the current used in the system:

$$P = I^2 * R$$
$$\text{power} = (\text{current})^2 * \text{resistance}$$
$$\text{Watts (W)} = (\text{Amps (A)})^2 * \text{Ohms } (\Omega)$$

Since your system is modeled as a resistor, if you focus on decreasing your current, you decrease your power consumption. Another way to look at power is as the product of voltage and current:

$$P = V * I$$
$$\text{power} = \text{voltage} * \text{current}$$
$$\text{Watts (W)} = \text{Volts (V)} * \text{Amps (A)}$$

This is why your processor core may run at 1.8V even though other parts of your system run at 3 or 5V. Since the core takes a lot of current, the lower voltage means less power consumption. The two ways of looking at power are equivalent, according to the golden rule of electrical engineering, *Ohm's law*:

$$V = I * R$$
$$\text{voltage} = \text{current} * \text{resistance}$$
$$\text{Volts (V)} = \text{Amps (A)} * \text{Ohms } (\Omega)$$

There is a lot more that Ohm's law can do for you, so if this overview has given you a buzz of interest, check out [“Further Reading” on page 296](#) for some electrical engineering reading.



One more useful equation is for energy:

$$E = P * T$$
$$\text{energy} = \text{power} * \text{time}$$
$$\text{Joules (J)} = \text{Watts (W)} * \text{seconds}$$

For the first time, we have an equation with time. Basically, it says that if you are designing an energy-efficient system, you can minimize the power it uses or the amount of time it is on.

Batteries have a voltage rating and a capacity. An alkaline AA battery has a nominal voltage of 1.5V and a capacity of about 3000mAh (milliamp hours). So capacity isn't rated in power but in current supplied over time. If your system takes 30mA and 1.5V when it is on, with an AA battery, it should last about 100 hours (~4 days). If your system takes 300mA, it should last about 10 hours. However, if your system consumes 3000 mA, the AA battery won't last an hour, because, just as with calling your system a resistor, this rule of thumb only goes so far. Your battery will have a datasheet that explains its capacity, peak current draw, and other characteristics.

From both perspectives (power and battery capacity), minimizing current consumption is the key to reducing power usage or increasing battery life.

Measuring Current

To keep score during your efforts to reduce power consumption, you'll need to know how to measure current consumption. If you have a bench top power supply, ignore what it says as they aren't very accurate.

Chapter 3 suggested that you get your own digital multimeter (DMM). If you get a good one, you can measure current directly as shown in part b of Figure 10-1. Most cheap DMMs will let you measure in the range of milliamps (mA). A small embedded system consisting of a low power processor and a few peripheral chips will draw tens to hundreds of milliamps. So the DMM is an easy way to measure current of a running system. However, if you are trying to look at power consumption in sleep mode, you may need to measure lower amounts of current. A good DMM may let you look at hundreds of microamps (uA) but you may need even finer granularity than that.

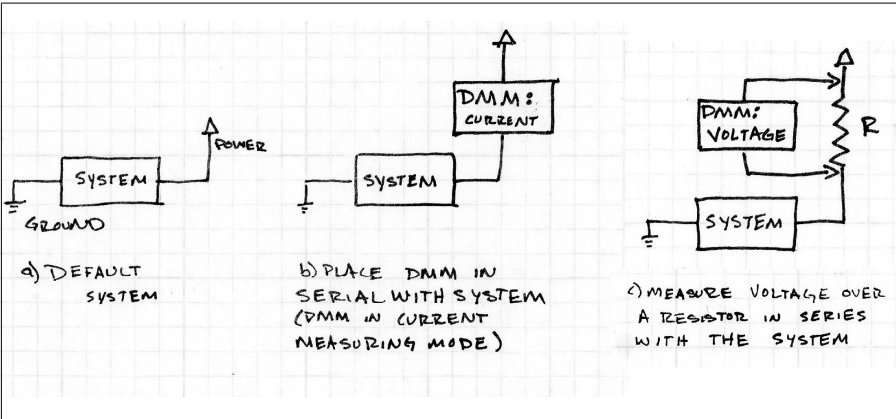


Figure 10-1. Measuring Current

Another way to measure current is to put a small resistor in series with the system as shown in part (c) of Figure 10-1. Switch your DMM to voltage mode, something it is better at measuring. You may need to tweak the value of the resistor to get the voltage in the range that your DMM can read it. Then apply Ohm's law:

$$\text{current} = \text{voltage} / \text{resistance}$$

This table shows some possible combinations to give you the idea. You may have to tweak the resistor value depending on the current you expect to read.

Expected Current	Resistor in Circuit	Voltage Measured on DMM	Note
12 mA	1 Ω	0.012 V	Easily measured on most DMMs
23 uA	1 Ω	0.000023 V	Impossible to read on most DMMs

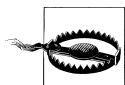
Expected Current	Resistor in Circuit	Voltage Measured on DMM	Note
34 μ A	10 Ω	0.00034 V	A very good DMM might be able to read this
45 μ A	100 Ω	0.0045 V	Once again within reach of most DMMs but resistor is getting to be too large.

The resistor likely needs to be large enough to read the tiny sleep currents but still small enough that it doesn't become a large consumer of power itself and interfere with the measurement. Since the it consumes power on its own, you may choose to have the resistor only available on development boards with a zero-ohm resistor (also known as a "wire") on the production boards.



Resistors are associated with an accuracy. After you measure current consumption, power off your system and measure the true resistance value of the resistor with your DMM.

Figure 10-1 shows how to measure the system current, but if you have multiple power rails (i.e. 5V, 3.3V, 1.8V), you may want to measure each individually or focus on the power rail most closely associated with the processor. The process is the same, though you might need a hardware engineer to set it up on your board.



Measuring the system current with your debugging tools attached might change the results.

(Thanks to Robert Mitchell for his excellent help on this sidebar!)

Turn Off the Light When You Leave the Room

The easiest way to reduce power consumption is turn off components that are not needed. The downside is that those components won't be ready when you need them, and bringing them back will add both some power usage and some latency in responding to events. You'll need to investigate the trade offs; I tend to use a spreadsheet to help me weigh the power savings.

The following sections details some of the ways to turn off or reduce the power drain of components.

Turn Off Peripherals

As you review your schematic and your design, consider what your processor has access to and how to design your system to turn off peripherals that are not needed. For

example, if you have external RAM, you may be able to preload the data to a local location and then power down the RAM instead of leaving it powered on for extended periods of time.

A chip external to the processor will consume no current if it does not have power. If the processor doesn't have access to the power lines, holding the peripheral in reset will nearly eliminate the current consumption. This isn't as optimal as turning off the power, but better than leaving the processor running.

Many chips need time to return to functioning after reset, so there is some overhead in powering them down. The goal is to turn off the component for a reasonable amount of time. ADCs are some of the peripherals with longer reset times. But they also often big power consumers, so it is often worth taking the time to balance the power consumption versus the power-on time.

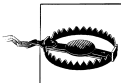
Turn Off Unused I/O devices

If you've got spare I/O devices, you can save a little bit of current by configuring them to be inputs with internal pull-downs. If your chip doesn't have internal pull-downs, set it to be a low output. If that doesn't work for you, an input with pull-ups still has reasonably small amounts of leakage current.



Don't remember pull-ups as anything other than gym torture? See [Chapter 3](#) for more information on both pull-ups and pull-downs.

As for those I/Os connected to the component you've got powered off, the preferred order is the same (input with pull-downs, output low, input with pull-ups). Configuring one I/O in this manner isn't likely to save a lot, but if you've got a whole bunch of them, the small savings can add up. On the other hand, if some I/Os on your system have an external pull-up or pull-down, set the internal one to match. Fighting external resistors wastes power.



If you have peripheral that is powered off (or in reset), be careful how you connect other processor I/Os to it. Don't leave the lines pulled-up or set high. Chips often have internal protection diodes that could use the high signal to back-power and damage the chip.

Turn Off Processor Subsystems

In addition to setting I/O devices to be low power, often you can turn off whole subsystems of the processor that you don't need (are you using that second SPI port?) Your processor manual will tell you whether this is a supported feature of the processor.

Sometimes you can turn off all functionality (best); sometimes you can only turn off the clock to the peripheral (still pretty good).

Slowing Down to Conserve Energy

Lowering your clock frequency saves power. This is true of all of the clocks you can control, but especially the processor clock. However, a slower processor clock gives you fewer processor cycles to run in. That is why we spent so much time in [Chapter 8](#) on optimizing code, even though optimizing for processor speed tends to decrease the quality (robustness, readability, and debuggability) of your code. Often, the most straightforward way to decrease power consumption of your system by 10% is to decrease the clock by about 10%. In other words, the power consumption of the chip is proportional to the frequency it runs at.

If the processor has to be on all the time, (for example, monitoring its environment), being able to slow down may be crucial to achieving low power. Some embedded operating systems will do frequency scaling for you, consuming power efficiently when possible and still having the speed available when you need it. You can do this without an operating system, instead of scaling over the whole range, you might want to create slow, medium and fast modes and change the performance level based on events. However, as with running a race, there is a balance between slowing the processor down and keeping it on for a longer time (being a tortoise) or sprinting for a shorter period of time and then sleeping (being a hare).

Putting the Processor to Sleep

Even at a slow clock speed, the processor is consuming power. Slowing down will help, but what if your code spends a lot of time waiting for things to happen?

Many processors designed for low-power systems can go into an energy-conserving sleep when they aren't needed. They use interrupts to tell the processor when to wake up.

On your computer, *sleep* is a standby mode where processing is off but the memory is still powered. This lets the operating system wake up quickly, exactly where it had left off. Alternatively, in *hibernation*, the contents of RAM are written to the disk (or other non-volatile memory). It takes longer to wake up from hibernation, but the system uses much less power while in the state (maybe almost no power, depending on the make of your computer).

Most embedded processors offer a similar range of sleep possibilities, with some processors having lots of granularity and others opting to have only one low power mode. In decreasing levels of power consumption, some sleep modes you might find in your processor manual include:

Slow down

Going beyond frequency scaling, some processors allow you to slow the clock down to hundreds of hertz.

Idle or sleep

This turns off the processor core but keeps enabled timers, peripherals and RAM alive. Any interrupt can return the processor to normal running.

Deep sleep or light hibernation

In addition to the processor core being disabled, some (possibly all) peripherals can be configured to turn off. Be careful not to turn off the subsystem that generates the interrupt that will wake up the processor.

Deep hibernation or power down

The processor chooses which peripherals to be turned off (usually almost all of them). RAM is usually left in an unstable state, but the processor registers are retained, so the system doesn't need initialization on boot. Usually, only a wakeup pin or a small subset of interrupts can restart the processor.

Power off

The processor does not retain any memory, and starts from a completely clean state.

Deeper sleep modes place the processor into a lower power state but take longer to wake from, increasing system latency. You'll need to spend some time with the user manual to determine how to achieve the lowest power level possible that will meet your product's needs.

The wake-up interrupts could be buttons, timers, other chips (i.e. ADC conversion complete), or traffic on a communications bus. It depends on your processor (and your sleep level).

Once you've determined that your processor can sleep, how do you design a system to make the best use of it? We'll explore that now.

Interrupt-based code flow model

Understanding the interrupt-based code flow model is critical for power sensitive applications. Breaking down the long name in the title of this section, *interrupt-based* indicates that the code will listen to interrupts to do its business. And if you have a subsystem that doesn't normally cause an interrupt, you'll need to make one, probably with a timer that can do all of the regular housekeeping. The next part of the name is *code flow* to show that the program tends to flow along like a waterfall. It doesn't have multiple tasks and it tends to be linear.

If you are familiar with an event loop for handling button presses in user interface code, the principle here is very similar. Button handlers tend to spend a lot of time idling, waiting for a button to be pressed. Once one is, the handler calls the appropriate function, and after it returns, the handler goes back to waiting for something to happen.

Instead of idling, the microprocessor spends time sleeping, waking up only when it needs to, then handling the interrupt and going back to sleep. The goal is to maximize the amount of time the processor is asleep, thereby maximizing power efficiency. See [Figure 10-2](#) for a flow chart describing the interrupt-based code flow model.

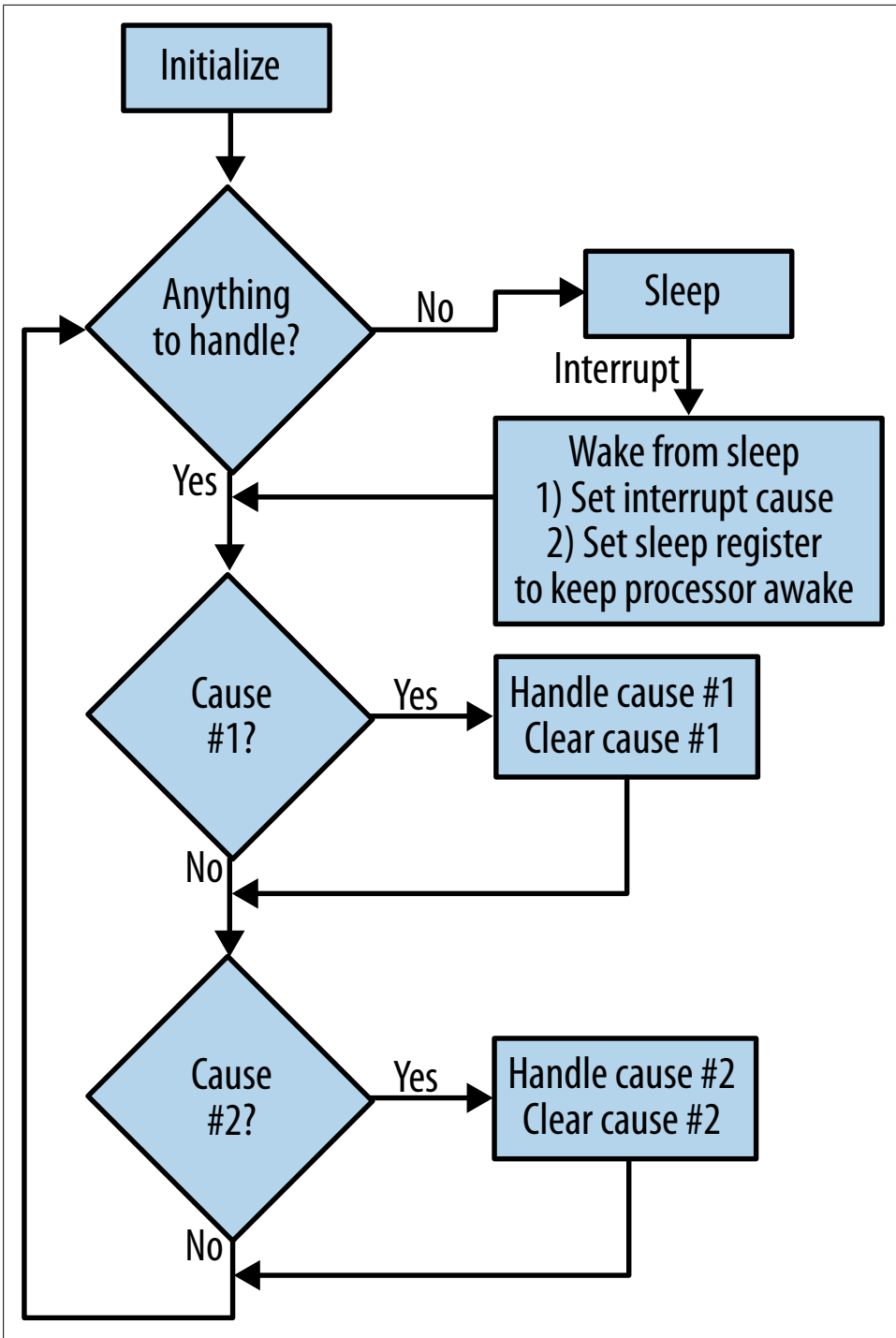


Figure 10-2. Interrupt-based code flowchart

The interrupt may be generated from a peripheral to indicate it needs processor attention (like an ADC finishing with its data, needing the processor to read the result and start another conversion). In other systems, the interrupt may be generated by a button the user presses or a sensor rising above a threshold. The interrupts may also be generated internally, for example, by a timer.

When an interrupt occurs, the processor wakes up and calls the appropriate interrupt service routine (ISR). The ISR sets a flag to indicate the cause for the wakeup and modifies the sleep register to keep the processor awake. When the ISR returns, because the *sleep register* is set to awake, the main loop runs. The main loop checks each possible flag and, when it finds a flag set, runs the appropriate handler (also clearing the flag). When the main loop completes and all of the flags have been cleared, the processor returns to sleep mode. [Figure 10-2](#) illustrates the general idea with two handlers to check.

Let's say you've got a processor that can sleep. To experiment, start with a timer interrupt. These are common in the interrupt flow model because the timer lets you wake up to do any necessary housekeeping. Run the processor without sleep, and set up an interrupt to run so it toggles an output line or LED. The frequency doesn't matter, though a slow interrupt is better. I'd recommend about twice per second (2Hz). Measure the current of the system as describe in [“Measuring Current” on page 287](#).

Now set your processor to sleep between interrupts. From the outside, the processor behaves the same. However, when you measure the system current, it should be smaller. How much smaller depends on many aspects of your system. For instance, I tried this procedure with the Texas Instrument MSP430 Launchpad development kit using the G2231 processor. When the system was in a while loop, waiting for the next timer interrupt, the system consumed 9mW (3mA @ 3V). When I let it enter a low power mode instead of idling in the while loop, the results were dramatically different. I used the third level of sleep (the processor has four). However, even with a 0.001 Ω resistor, I couldn't measure any voltage drop over the resistor with my DMM.

The datasheet informed me that the current should be about 0.9uA (2.7uW) in this state, which would require additional special tools for me to measure. The timer interrupt to toggle the LED must have spent some time at 3mA, but the processor woke up, set the timer flag, toggled the LED in main, and went back to sleep before it could make a blip on my meter. Remember, *energy efficiency* is about time as well as current.

The processor on TI's Launchpad kit is designed to be super low power when it is sleeping, so the results may not be so dramatic on another device.

A Closer Look at the Main Loop

Handling the interrupts in the main loop allows you to keep the interrupt service routines short, because all they do is set a couple flags and exit. Long interrupts tend to decrease system responsiveness. Plus, going through the main loop can aid in debugging

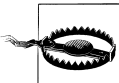
as the system revolves around sleeping and the flags set by the most recent interrupts. (If you can output the cause variable on every pass, you'll get a good view of what the system is doing.) Note that additional interrupts may happen while executing the handlers for the interrupt that caused the processor to wake up.

By putting the flags in the bits of a variable (or two) it becomes simple to check that all interrupts have been processed.

```
volatile uint16_t gInterruptCause = 0; // Bitmasked flags that describe what interrupt has occurred

void main()
{
    uint16_t cause;
    Initialize();
    while (1) {
        cause = gInterruptCause; // atomic interrupt safe read of gInterruptCause
        while (cause) {
            // each handler will clear the relevant bit in gInterruptCause
            if (cause & 0x01) { HandleSlowTimer(); } /
            if (cause & 0x02) { HandleADCCConversion(); }
            FeedWatchdog(); // while awake, make sure the watchdog stays happy
            cause = gInterruptCause;
        }
        // need to read the cause register again without allowing any new interrupts:
        InterruptsOff();
        if (gInterruptCause == 0) {
            InterruptsOn();
            GoToSleep(); // an interrupt will cause a wake up and run the while loop
        }
        InterruptsOn();
    }
} // end main
```

The interrupts themselves just need to set the flag and modify the sleep register to keep the CPU awake after they return. The handlers will perform the necessary actions and clear the cause register (in an interrupt safe manner).



Remember that working with interrupts can be tricky due to race conditions. [“Interrupts” on page 123](#) for more tips on interrupts.

Processor Watchdog

In [“Watchdog” on page 143](#), I talked about the importance of having a watchdog, and how you shouldn't use a timer interrupt to service the watchdog. Now that your system is entirely interrupt based, how are you going to make sure the watchdog remains content?

In the lightest levels of sleep, your processor will probably let the watchdog continue to run. The processor may let you configure whether you want the watchdog to run while the processor is asleep (consuming some power during sleep and forcing the

processor to wake up to service it) or to have the watchdog sleep when the processor does (losing a failsafe mechanism for your system). Recognizing the trade off will help you make the right decision for your system.

If you leave the watchdog on, you'll need to set a timer to wake the processor up before the watchdog expires. This goes against the grain of my earlier advice. However, here the watchdog is not only verifying that the system is running properly, it is verifying that the system is waking up from sleep properly.

Avoid Frequent Wake-ups

Since each wakeup requires some overhead from the chip (and the deeper the sleep, the more overhead is required as it wakes up), avoid frequent wake-ups. If you have a low-priority task that doesn't have a hard real-time requirement, piggyback it upon another task. This spreads the overhead of waking up the processor over several tasks.

While a timer to do housekeeping can be valuable, it may be better to check whether enough time has passed to do the low priority tasks and reset the timer so you can skip a wakeup.

Chained Processors

It is pretty common to have a small, very efficient processor monitor the important signals, waking up a larger processor when needed. In this case the larger processor requires more power to wake up (and possibly more time if it is running an operating system). So the small processor does the housekeeping: checking for buttons pressed, looking for lower power conditions that indicate the system should shut down, or waking the large processor due to a preset alarm.

In such systems, both processors spend as much time asleep as possible, but the system is designed so that the interrupts get triaged in the small processor, which can opt to handle them itself or wake up its larger partner.

Further Reading

As noted in [Chapter 3](#), for a gentle introduction to electronics, I recommend *Make: Electronics* by Charles Platt, published by O'Reilly (2009). The step by step introduction to putting a system together and soldering is an easy read.

For a more serious look at electrical engineering, *The Art of Electronics* by Paul Horowitz and Winfield Hill (Cambridge University Press, 1989) is the seminal book, along the lines of Knuth's *Art of Computer Programming*.

If you are somewhere in between those two and looking for a more intuitive grasp of analog components, try *There Are No Electrons: Electronics for Earthlings* by Kenn Amdahl (Clearwater Publishing, 1991). This fun book gets away from the plumbing

model of electronics and gives you a different story about how resistors and capacitors work.

The TI MSP430 line of processors is geared toward inexpensive and power sensitive devices. The [MSP430 Software Coding Techniques Application Report](#) is an excellent reference for learning more about the interrupt flow model and other power saving tips.

Interview question: Is the fridge light on?

In two minutes, how many different ways can you figure out how to tell whether the light in a fridge is going off when you shut the door? Please don't damage the fridge. [Thanks to Jen Silva for this question.]

This question is all about quantity and creativity of solutions offered. This is not a question with a lot of depth (though I will ask a follow-up if I want more depth). Lots of people come up with the common solutions, some of which are:

- Take a video recording
- Wireless (and presumably battery powered) light sensor
- Solar panel charger (check the charge of a battery after a period of time, with and without the door open)
- Disabling the cooling assembly so the fridge uses only a small amount of power, then checking the difference in current with the door open and closed
- With the door open, press the door closing button to see what happens

Interviewees get credit for the number of solutions, uniqueness of them, what implicit assumptions they make, and how they apply scientific method to solve the problem. I like to see them note the trade-offs in terms of time and cost but I'd rather they spent the time generating different answers.

There seem to be three types of responses when faced with this question. First (and worst) are the interviewees that come up with one of these and can't figure something else out, even with prompting. Second (and most common) are the interviewees that come up with a solution and tweak one thing at a time until they get to another solution. They generally rack up enough solutions and often surprise me with a few really creative ones. The last set of responses can only be classified as mildly insane. These are the most fun and, as long as they manage to get a few of the more sensible answers, these are the best people to work with.

One person suggested a statistical method in answer: first you get 100 units. Leave the door open on 50 of them and the door closed on 50. Determine the mean time between failure (MTBF) for the light bulbs in the open units. Now open all the closed doors and wait for those to fail. If they fail about the same time as the initially opened ones, then the light was off with the door closed.

This isn't necessarily sensible or easily accomplished in a reasonable amount of time for testing the lights in fridges, but there are times when this sort of method will work very well. It gets bonus points for extra creativity.

