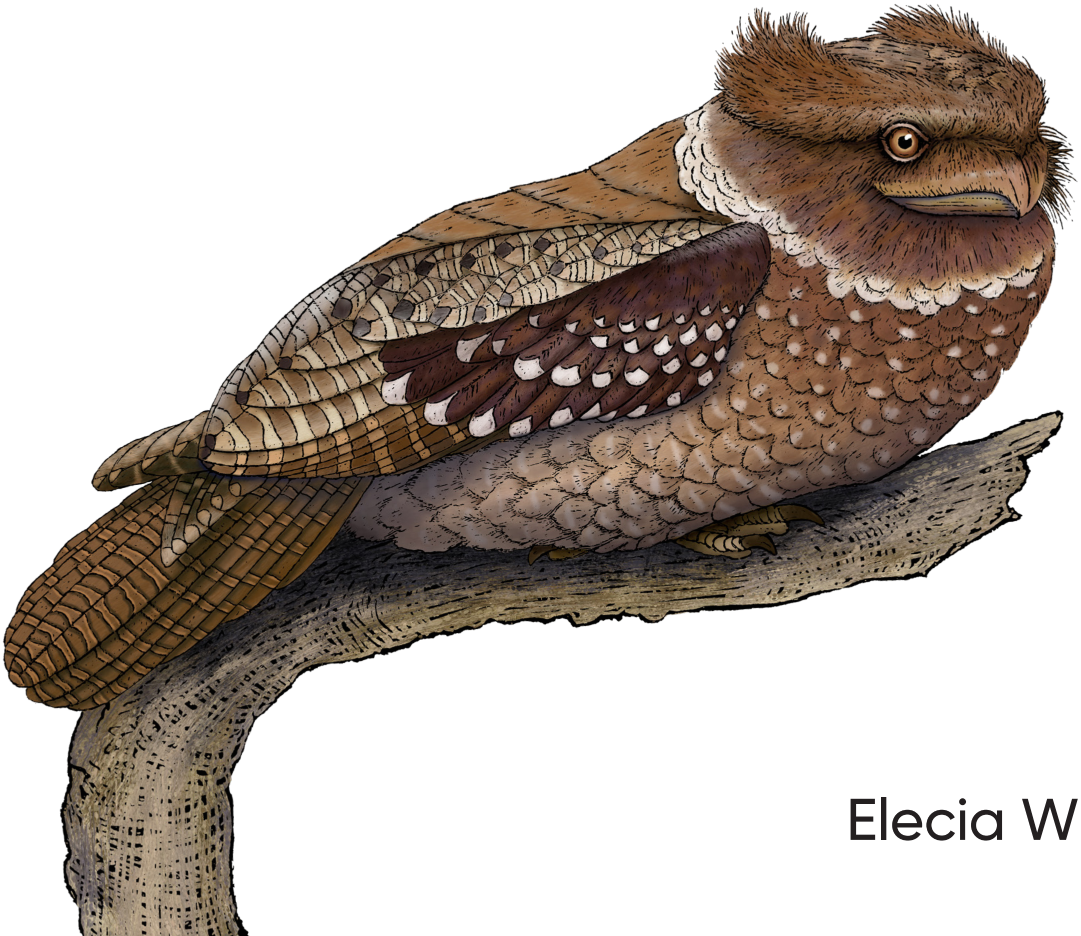
O'REILLY®

Second Edition

Making Embedded Systems

Design Patterns for Great Software



Elecia White

O'REILLY®

Making Embedded Systems

Interested in developing embedded systems? Since they don't tolerate inefficiency, these systems require a disciplined approach to programming. This easy-to-read guide helps you cultivate good development practices based on classic software design patterns and new patterns unique to embedded programming. You'll learn how to build system architecture for processors, not for operating systems, and you'll discover techniques for dealing with hardware difficulties, changing designs, and manufacturing requirements.

Written by an expert who has created systems ranging from DNA scanners to children's toys, this book is ideal for intermediate and experienced programmers, no matter what platform you use. This expanded second edition includes new chapters on IoT and networked sensors, motors and movement, debugging, data handling strategies, and more.

- **Optimize your system to reduce cost and increase performance**
- **Develop an architecture that makes your software robust in resource-constrained environments**
- **Explore sensors, displays, motors, and other I/O devices**
- **Reduce RAM, code space, processor cycles, and power consumption**
- **Learn how to interpret schematics, datasheets, and power requirements**
- **Discover how to implement complex mathematics on small processors**
- **Design effective embedded systems for IoT and networked sensors**

"Elecia White did it again! This second edition of her highly acclaimed book makes the fascinating subject of embedded software development approachable and fun. It covers all the essential topics necessary to orient newcomers in the intricacies of embedded development processes, patterns, and tools."

—Miro Samek
Embedded systems expert, author, and teacher

Elecia White is the founder and a principal embedded systems engineer at Logical Elegance, Inc. She's worked on ICU monitors, inertial measurement units for airplanes and race cars, educational toys, and assorted other medical, scientific, and consumer devices. A graduate of Harvey Mudd College in Claremont, California, Elecia is also cohost of *Embedded.FM*, a podcast about embedded systems, engineering careers, and creative technology.

EMBEDDED SYSTEMS

linkedin.com/company/oreilly-media
youtube.com/oreillymedia

US $49.99 CAN $62.99
ISBN: 978-1-098-15154-6

54999

9 781098 151546

SECOND EDITION

Making Embedded Systems

Design Patterns for Great Software

Elecia White

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Making Embedded Systems
by Elecia White

Copyright © 2024 Elecia White. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editor: Brian Guerin
Development Editor: Shira Evans
Production Editor: Katherine Tozer
Copyeditor: Penelope Perkins
Proofreader: Emily Wydeven
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

October 2011:	First Edition
March 2024:	Second Edition

Revision History for the Second Edition

2024-03-01: First Release

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

978-1-098-15154-6

LSI

Table of Contents

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.[1] 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. 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, and children's toys, I've found a lot of commonalities. There are 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 successful 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 infer how you think. The most useful 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

1 Erich Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley).

interviewer might look for in your response. You'll have to get the job (and the answers) on your own merits.

One note, though: my embedded systems don't have operating systems (OS). 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 working with an embedded 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.

Who This Book Is For

I wrote this for some folks I've worked with in the past.

Sarah was a junior software engineer who joined my embedded systems team. She was bright and interested but didn't know how to handle hardware.

Josh was an experienced electromechanical engineer who needed to write software. He could power through some code but got stuck on designing the system, debugging memory issues, and reusing code.

Usually we only learn software or hardware in school; we don't learn how to make them work together. My goal is to cantilever off the knowledge you have to fill the gaps.

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, scientific, 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.

After the first edition of this book, I started the Embedded.fm podcast to talk about embedded systems with other people. Through hundreds of episodes, I've learned how other engineers solve problems, about new technologies being developed, and other career paths.

Organization of This 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 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 that fire.

The order of chapters is:

Chapter 1, "Introduction"
: This chapter describes what an embedded system is and how development is different from traditional software.

Chapter 2, "Creating a System Architecture"
: Whether you are trying to understand a system or creating one from scratch, there are tools to help.

Chapter 3, "Getting Your Hands on the Hardware"
: Hardware/software integration during board bring-up can be scary, but there are some ways to make it smoother.

Chapter 4, "Inputs, Outputs, 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, "Interrupts"
: Interrupts are one of the most confusing topics in embedded systems: code that gets called asynchronously on events that happen inside the processor. A chicken is used to make this easier.

Chapter 6, "Managing the Flow of Activity"
: This chapter describes methods for setting up the main loop of your system, where to use interrupts (and how not to), and how to make a state machine.

Chapter 7, "Communicating with Peripherals"
: Different serial communication methods rule embedded systems: UART, SPI, I2C, USB, and so on. While you can look up the details for each, this chapter looks at what makes them different from each other and how to make them work more efficiently.

Chapter 8, "Putting Together a System"
: Common peripherals such as LCDs, ADCs, flash memory, and digital sensors have common implementation needs such as buffer handling, bandwidth requirements, and pipelines.

Chapter 9, "Getting into Trouble"
: Debugging is a skill every developer needs. Figuring out how to cause problems will teach you how to solve bugs, stack problems, hard faults, and cleverness.

Chapter 10, "Building Connected Devices"
: Whether you have consumer IoT devices or industrial networked systems, managing many devices means dealing with firmware updates, security, and monitoring health.

Chapter 11, "Doing More with Less"
: Optimization is not for the faint of heart. This chapter shows methods for reducing consumption of RAM, code space, and processor cycles.

Chapter 12, "Math"
: Most embedded systems need to do some form of analysis. Understanding how mathematical operations and floating points work (and don't work) will make your system faster and more robust.

Chapter 13, "Reducing Power Consumption"
: From reducing processor cycles to system architecture suggestions, this chapter will help you if your system runs on batteries.

Chapter 14, "Motors and Movement"
: This chapter is a basic introduction to motors and movement. (Or possibly the introduction to an entirely new book.)

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, considering that most people don't get to it until later in their career. But I want folks to think about how their code fits in the system long before I want them to worry about optimization.

Terminology

A *microcontroller* is a processor with onboard goodies like RAM, code space (usually flash), and various peripheral interfaces (such as I/O lines). Your code runs on a processor, or *central processing unit* (CPU). A *microprocessor* is a small processor, but the definition of "small" varies.

A DSP (*digital signal processor*) is a specialized form of microcontroller that 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, multiplication and addition).

As I wrote this book, I wanted to use the correct terminology so you'd get used to it. However, with so many names for the piece of the system that is running your code, I didn't want to add confusion by changing the name. So, I stick with the term *processor* to represent whatever it is you're using to implement your system. Most of the material is applicable to whatever you actually have.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, 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, databases, data types, environment variables, statements, and keywords.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

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.

This book has a GitHub repository for code, tools, and pointers to more information.

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 2024 Elecia White, 978-1-098-15154-6."

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*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.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
> 800-889-8969 (in the United States or Canada)
> 707-827-7019 (international or local)
> 707-829-0104 (fax)
> *support@oreilly.com*
> *https://www.oreilly.com/about/contact.html*

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/making-embedded-systems-2*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

This book didn't happen in a vacuum. It started with a colleague who said, "Hey, do you know a book I can give to one of my junior engineers?" From that planted seed came months of writing; I learned to really appreciate understanding (and encouraging) friends. Then there were the engineers who gave their time to look at the technical material (any remaining issues are my fault, not theirs, of course). Finally, O'Reilly provided tremendous support through the whole process.

Thanking each person as they properly deserve would take pages and pages, so I will just go through them all in one breath, in no particular order: Phillip King, Ken Brown, Jerry Ryle, Matthew Hughes, Eric Angell, Scott Fitzgerald, John Catsoulis, Robert P. J. Day, Rebecca Demarest, Jen Costillo, Phillip Johnston, Rene Xoese Kwasi Novor, and Chris Svec. These folks made a difference in the flavor of this book. There are additional thank-yous spread throughout the book, where I got help from a particular person in a particular area, so you may see these names again (or a few different ones).

Finally, authors always give gushing thanks to their spouses; it is a cliché. However, having written a book, I see why. Christopher White, my favorite drummer, physicist, and embedded systems engineer, thank you most of all. For everything.

CHAPTER 1
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 nontechnical people that embedded systems are things like microwaves and automobiles that run software but aren't computers. (Most people recognize a computer as a general-purpose device.) Perhaps an easy way to define the term without haggling over technology is:

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

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

The hardware isn't the only part of the system with constraints. In some systems, the software must act deterministically (exactly the same each time) or in real time (always reacting to an event fast enough). Some systems require that the software be fault-tolerant, with graceful degradation in the face of errors. For example, consider a system in which servicing faulty software or broken hardware may be infeasible (such as a satellite or a tracking tag on a whale). Other systems require that the software cease operation at the first sign of trouble, often providing clear error messages (for example, a heart monitor should not fail quietly).

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

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

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

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

Embedded Systems Development

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

Compilers and Languages

Embedded systems use *cross-compilers*. Although a cross-compiler runs on your desktop or laptop computer, it creates code that does not. The cross-compiled image runs on your target embedded system. Because the code needs to run on your embedded 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 such as GCC.

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

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

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

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

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

Debugging

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

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

The processor must expend some of its resources to support the debug interface, allowing the debugger to halt it as it runs and providing the normal sorts of debug information. Supporting debugging operations adds cost to the processor. To keep costs down, some processors support a limited subset of features. For example, adding a breakpoint causes the processor to modify the memory-loaded code to say "stop here." However, if your code is executing out of flash (or any other sort of read-only memory), instead of modifying the code, the processor has to set an internal register (hardware breakpoint) and compare it at each execution cycle to the code address being run, stopping when they match. This can change the timing of the code, leading to annoying bugs that occur only when you are (or maybe aren't) debugging. Internal registers take up resources, too, so often there are only a limited number of hardware breakpoints available (frequently there are only two).

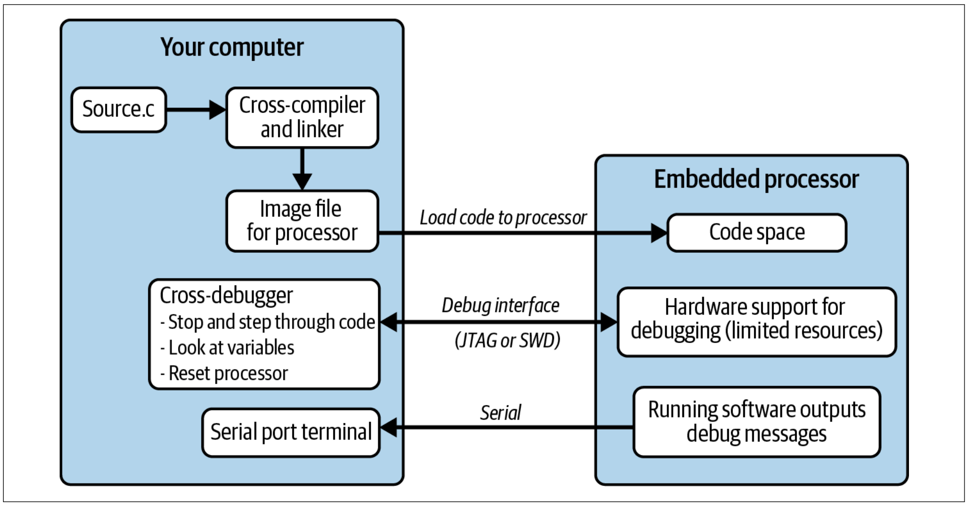


Figure 1-1. Computer and target processor

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

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

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

Writing software for an embedded system can be tricky, as you have to balance the needs of the system and the constraints of the hardware. Now you'll need to add another item to your to-do list: making the software debuggable in a somewhat hostile environment, something we'll talk more about in Chapter 2.

Resource Constraints

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

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

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

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

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

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

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

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

Principles to Confront Those Challenges

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

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

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

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

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

You don't have to wait for someone else to give you presents. Testing and quality go hand in hand. As you are thinking about how to write a piece of code, spend some time considering how you will test it. Writing test code for your system will make it better, provide some documentation for your code, and make other people think you write great software.

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

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

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

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

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

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

Prototypes and Maker Boards

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

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

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

Software aside, when the existing off-the-shelf boards don't meet your needs, custom boards are often necessary. The development boards may be too delicate, connected by fragile wires. They may be too expensive for your target market or take too much power. They may be the wrong size or shape. They may not hold up under the intended environmental conditions (like temperature fluctuations in a car, getting wet, or going to space).

Whatever the reason for getting a custom board, it usually means removing the programming (and debugging) hardware that are part of processor development boards.

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

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

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

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

Further Reading

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

- First is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, et al. (Addison-Wesley). Originally published in 1995, this book sparked the software design pattern revolution. Due to its four collaborators, it is often known as the "Gang of Four" book (or a standard design pattern may be noted as a GoF pattern).
- Second is *Head First Design Patterns* by Eric T. Freeman, et al. (O'Reilly). This book is far more readable than the original GoF book. The concrete examples were much easier for me to remember.

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

Interview Question: Hello World

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

In many embedded system projects, you have to develop a system from scratch. With the first part of this task, I look for a candidate who can take a blank slate and fill in the basic functionality, even in an unfamiliar environment. I want them 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 they 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 their beautiful algorithms. When asked what happens before the main function, they will tend to answer along the lines of "you know, the program runs," but with no understanding of what that implies.

However, if they mention "start" or "cstart," they are well on their way in the interview. In general, I want them to know that the program requires initialization beyond what we see in the source, no matter what the platform is. I like to hear mention of setting the exception vectors to handle interrupts, initializing critical peripherals, initializing the stack, initializing variables, and if there are any C++ objects, calling constructors for those. It is great if they 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 these steps are important and how they happen in an embedded system. An experienced embedded engineer often starts at the vector table, with the reset vector, and moves from there to the power-on behavior of the system. This material is covered later in the book, so if these terms are new to you, don't worry.

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

CHAPTER 2

Creating a System Architecture

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

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

In this chapter, we'll look at different ways of viewing the system in an effort to design better embedded software. We'll create a few diagrams to describe what our system looks like and to identify how we should architect our software. In addition to better understanding the system, the goal of the diagrams is to identify ways to architect the software to changing requirements and hardware: encapsulation, information hiding, having good interfaces between modules, and so on.

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

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

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

Getting Started

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

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

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

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

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

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

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

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

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

Creating System Diagrams

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

I recommend four types of diagrams: context diagrams, block diagrams, organigrams, and layering diagrams.

The Context Diagram

The first drawing is an overview of how your system fits into the world at large. If it is a children's toy, the diagram may be easy: a stick figure and some buttons on the toy. If it is a device providing information to a network of seismic sensors with an additional output to local scientists, the system is more complicated, going beyond the piece you are working on. See Figure 2-1.

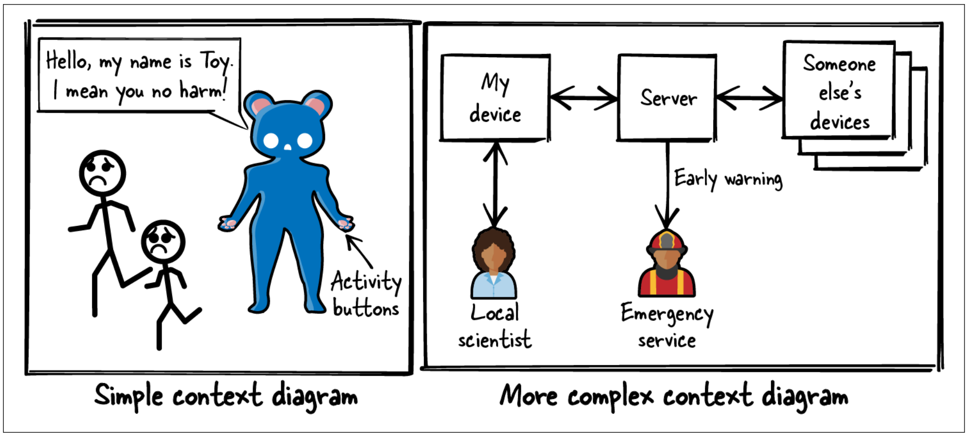


Figure 2-1. Two context diagrams with different levels of complexity

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

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

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

The Block Diagram

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

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

For this example, I'm going to introduce a type of memory called *flash memory*. While the details aren't important here, it is a relatively inexpensive type of memory used in many devices. Many flash memory chips communicate over the SPI (Serial Peripheral Interface, usually pronounced "spy") bus, a type of serial communication (discussed in more detail in Chapter 7). Most processors cannot execute code over SPI, so the flash is used for off-processor storage. Our schematic, shown at the top of Figure 2-2, shows that we have some flash memory attached to our processor via SPI. Don't be intimidated by the schematic! We'll cover them in more detail in Chapter 3.

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

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

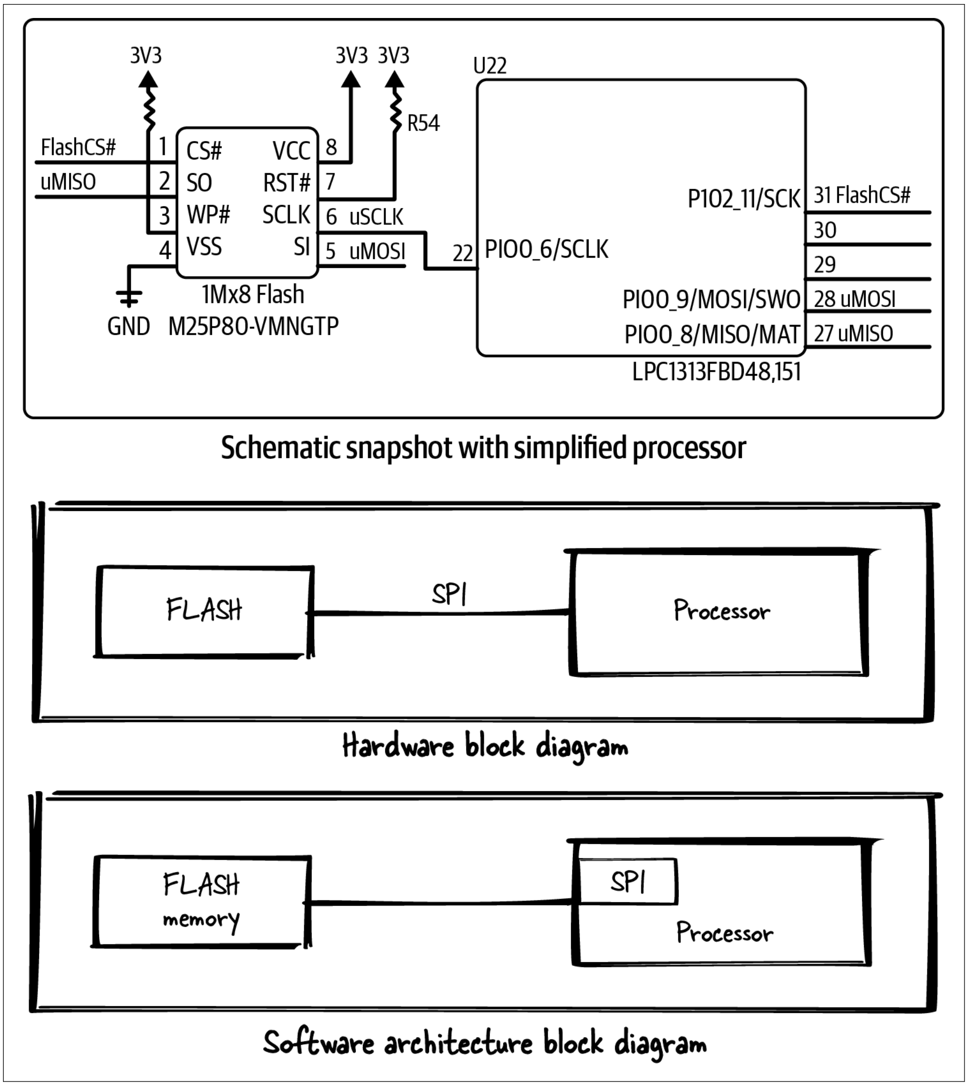


Figure 2-2. Comparison of schematic and initial hardware and software block diagrams

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

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

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

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

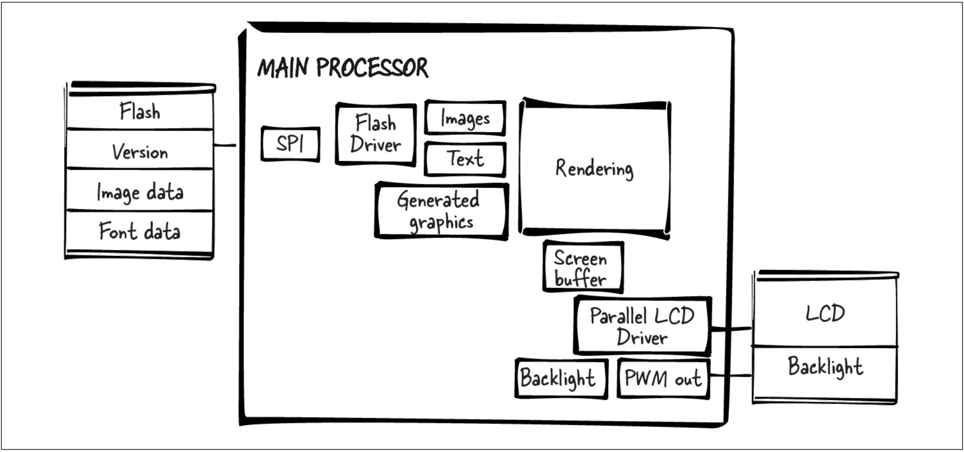


Figure 2-3. More detailed software block diagram

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

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

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

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

Organigram

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

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

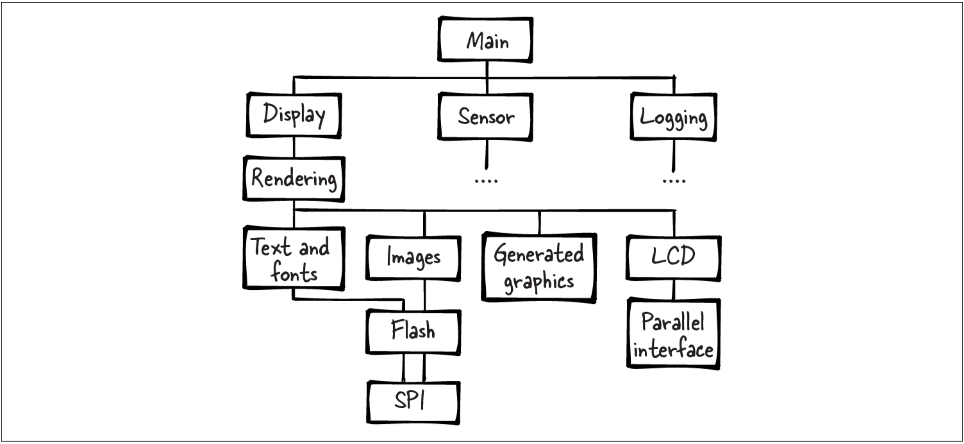


Figure 2-4. Organizational diagram of software architecture

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

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

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

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

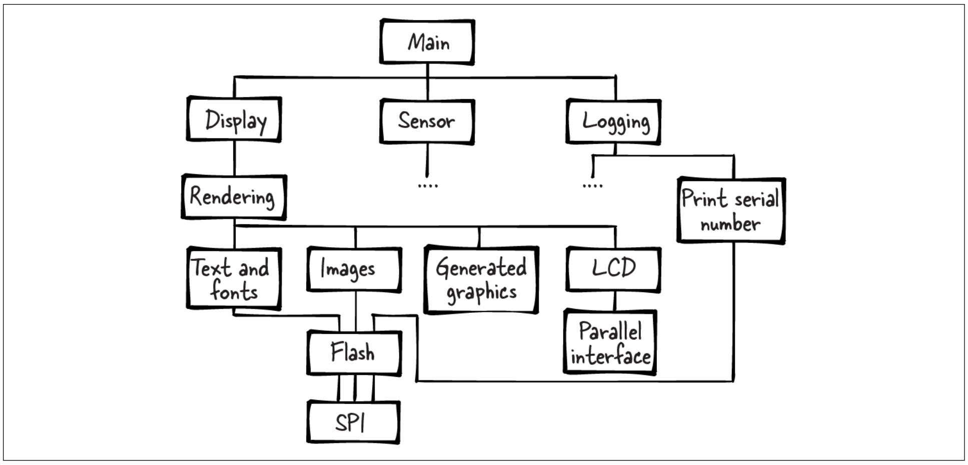


Figure 2-5. Organizational diagram with a shared resource

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

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

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

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

Layering Diagram

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

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

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

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

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

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

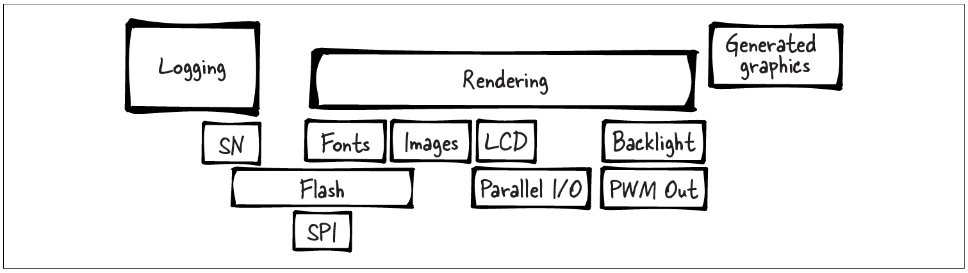


Figure 2-6. Software architecture layering diagram

Designing for Change

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

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

Encapsulate Modules

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

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

Here are some things to look for:

- In the organigram, look for objects that are used by only one other object. Are these both fixed? If they are unlikely to change or are likely to change together as you evolve the system, consider combining them. If they change independently, it is not a good area to consider for encapsulation.
- In the layering diagram, look for collections of objects that are always used together. Can these be grouped together in a higher-level interface to manage the objects? You'd be creating a hardware abstraction layer.
- Which modules have lots of interdependencies? Can they be broken apart and simplified? Or can the dependencies be grouped together?
- Can the interfaces between vertical neighbors be described in a few sentences? That makes it a good place for encapsulation to help you create an interface (for others to reuse your code or just for ease of testing).

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

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

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

Delegation of Tasks

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

Too often we want our minions to do the boring part of the job while we get to do all of the fun parts. ("Here, minion, write my test code, do my documentation, fold my laundry.") Not only does this drive away the good minions, it tends to decrease the quality of your product. Instead, think about which whole box (or whole subtree) you can give to someone else. As you try to describe the interdependencies to your imaginary minion, you may find that they become worse than you've represented in the diagrams. Or you may find that a simple flag (such as a semaphore) to describe who currently owns the resource may be enough to get started.

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

Looking for things that can be split off and accomplished by another person will help you identify sections of code that can have a simple interface between them. Also, when marketing asks how this project could be made to go faster, you'll have an answer ready for them. However, there is one more thing our imaginary minion provides: assume they are 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).

Top-down design is when you think about what you want and then dig into what you need to accomplish your goals. *Bottom-up design* is when you consider what you have and build what you can out of that.

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

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