

Design Patterns for Great Software



Making Embedded Systems

O'REILLY®

Elecia White

Making Embedded Systems

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

Written by an expert who's created embedded systems ranging from urban surveillance and DNA scanners to children's toys, this book is ideal for intermediate and experienced programmers, no matter what platform you use.

- Optimize your system to reduce cost and increase performance
- Develop an architecture that makes your software robust in resource-constrained environments
- Explore sensors, motors, and other I/O devices
- Do more with less: reduce RAM consumption, code space, processor cycles, and power consumption
- Learn how to update embedded code directly in the processor
- Discover how to implement complex mathematics on small processors
- Understand what interviewers look for when you apply for an embedded systems job

Elecia White is the founder of Logical Elegance, an embedded systems consulting company based in San Jose. She enjoys making interesting gizmos, including medical devices, inertial measurement units, educational toys, and a gunshot location system.

“Making Embedded Systems is the book for a C programmer who wants to enter the fun (and lucrative) world of embedded systems. It's very well written—entertaining, even—and filled with clear illustrations.”

—Jack Ganssle
author and embedded
system expert

US \$39.99

CAN \$41.99

ISBN: 978-1-449-30214-6



9



Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY[®]
oreilly.com

Making Embedded Systems

Elecia White

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Making Embedded Systems

by Elecia White

Copyright © 2012 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://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editors: Andy Oram and Mike Hendrickson

Production Editor: Holly Bauer

Copyeditor: Genevieve d'Entremont

Proofreader: Kiel Van Horn

Indexer: Ellen Troutman Zaig

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

October 2011: First Edition.

Revision History for the First Edition:

2011-10-24 First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449302146> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Making Embedded Systems*, the image of a great-eared goatsucker, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-30214-6

[LSI]

1319490913

Table of Contents

Preface	ix
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	14
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	20
Example: A Logging Interface	21
A Sandbox to Play In	27
Further Reading	32
3. Getting Your Hands on the Hardware	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	45
Your Processor Is a Language	48
Reading a Schematic	50
Having a Debugging Toolbox (and a Fire Extinguisher)	53
Keep Your Board Safe	53
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
Dealing with Errors	69
Consistent Methodology	70
Error-Handling Library	70
Debugging Timing Errors	71
Further Reading	72
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	80
Separating the Hardware from the Action	82
Board-Specific Header File	82
I/O-Handling Code	83
Main Loop	85
Facade Pattern	86
The Input in I/O	87
A Simple Interface to a Button	88
Momentary Button Press	90
Interrupt on a Button Press	90
Configuring the Interrupt	91
Debouncing Switches	92
Runtime Uncertainty	94
Dependency Injection	95
Using a Timer	96
Timer Pieces	97
Doing the Math	99

A Long Wait Between Timer Ticks	103
Using the Timer	104
Using Pulse-Width Modulation	104
Shipping the Product	106
Further Reading	108
5. Managing the Flow of Activity	109
Scheduling and Operating System Basics	109
Tasks	109
Communication Between Tasks	110
Avoiding Race Conditions	110
Priority Inversion	112
State Machines	113
State Machine Example: Stoplight Controller	114
State-Centric State Machine	115
State-Centric State Machine with Hidden Transitions	115
Event-Centric State Machine	116
State Pattern	117
Table-Driven State Machine	118
Choosing a State Machine Implementation	121
Interrupts	121
An IRQ Happens	122
Save the Context	129
Get the ISR from the Vector Table	131
Calling the ISR	133
Restore the Context	136
When to Use Interrupts	136
How Not to Use Interrupts	137
Polling	138
System Tick	138
Time-Based Events	140
A Very Small Scheduler	141
Watchdog	142
Further Reading	144
6. Communicating with Peripherals	147
The Wide Reach of Peripherals	147
External Memory	147
Buttons and Key Matrices	148
Sensors	150
Actuators	153
Displays	158
So Many Ways of Communicating	163

Serial	165
Parallel	173
Ethernet and WiFi	175
Putting Peripherals and Communication Together	176
Data Handling	176
Adding Robustness to the Communication	186
Changing Data	189
Changing Algorithms	191
Further Reading	193
7. Updating Code	197
Onboard Bootloader	198
Build Your Own Updater	199
Modifying the Resident Updater	201
Brick Loader	202
Copy Loader to RAM	203
Run the Loader	204
Copy New Code to Scratch	205
Dangerous Time: Erase and Program	205
Reset to New Code	205
Security	206
Linker Scripts	207
Summary	210
8. Doing More with Less	213
Code Space	214
Reading a Map File (Part 1)	214
Process of Elimination	217
Libraries	218
Functions and Macros	219
Constants and Strings	220
RAM	221
Remove malloc	221
Reading a Map File (Part 2)	223
Registers and Local Variables	224
Function Chains	226
Pros and Cons of Globals	228
Memory Overlays	228
Speed	229
Profiling	230
Optimizing	234
Summary	243
Further Reading	244

9. Math	247
Identifying Fast and Slow Operations	248
Taking an Average	249
Use an Existing Algorithm	252
Designing and Modifying Algorithms	255
Factor Polynomials	255
Taylor Series	256
Dividing by a Constant	258
Scaling the Input	259
Lookup Tables	260
Fake Floating-Point Numbers	267
Rational Numbers	268
Precision	269
Addition (and Subtraction)	270
Multiplication (and Division)	271
Determining the Error	272
Further Reading	276
10. Reducing Power Consumption	279
Understanding Power Consumption	280
Turn Off the Light When You Leave the Room	282
Turn Off Peripherals	282
Turn Off Unused I/O devices	283
Turn Off Processor Subsystems	283
Slowing Down to Conserve Energy	284
Putting the Processor to Sleep	285
Interrupt-Based Code Flow Model	286
A Closer Look at the Main Loop	288
Processor Watchdog	289
Avoid Frequent Wake-Ups	290
Chained Processors	290
Further Reading	290
Index	293

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*. Boston: Addison-Wesley.

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.

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, and Jen Costillo. 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 on a particular area, so you may see these names again (or a few different ones).

Two people are left out of that list. Andy Oram is an excellent editor at O'Reilly, and I was lucky to get him. He's made a big difference to this book and to my writing.

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.

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

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 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, 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, Managing the Flow of Activity

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 ranging from internal updaters 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 points 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, considering that 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.



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



This icon indicates a warning or caution.

Terminology

A *microcontroller* is a processor with onboard goodies like RAM, code space (usually flash), and various peripheral interfaces (e.g., 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” changes.

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, multiply and add).

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.

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 2012 Elecia White, 978-1-449-30214-6.”

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.

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://shop.oreilly.com/product/0636920017776.do>

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

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>

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 (e.g., 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).

Compilers, Languages, and Object-Oriented Programming

Another way to identify embedded systems is that they use cross-compilers. Although a cross-compiler runs on your desktop or laptop computer, it creates code that does not. The cross-compiled image runs on your target embedded system. Because the code needs to run on your processor, the vendor for the target system usually sells a

cross-compiler or provides a list of available cross-compilers to choose from. Many larger processors use the cross-compilers from the GNU family of tools.

Embedded software compilers often support only C, or C and C++. In addition, many embedded C++ compilers implement only a subset of the language (multiple inheritance, exceptions, and templates are commonly 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"), regardless of whether it actually implements that widespread standard.

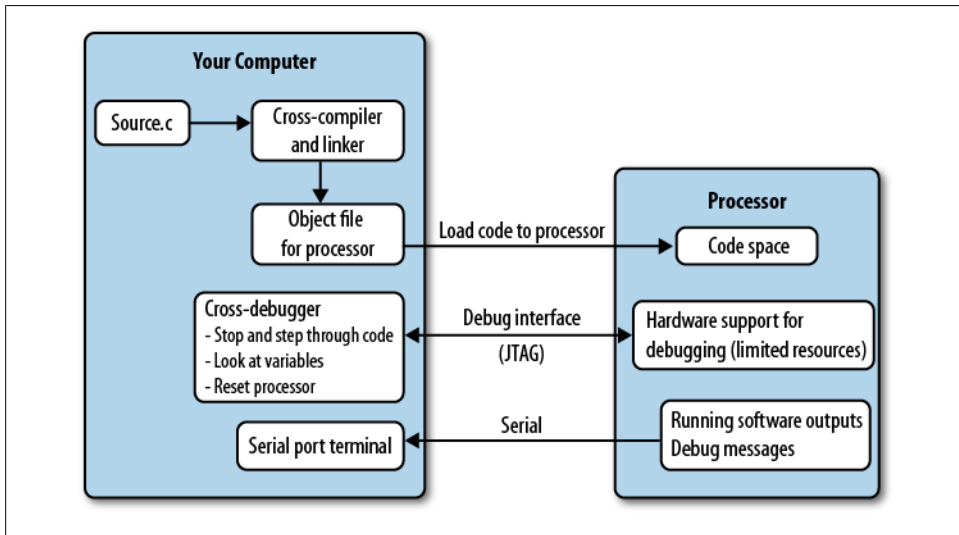


Figure 1-1. Computer and target processor

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

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

The device that communicates between your PC and the processor is generally called 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 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.

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 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 be challenged 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 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, but the needs of the project also change through its lifespan. In the beginning, maybe you want to hack something together just to try it out. As you get more serious and better understand (and define) the goals of the product and the hardware you are using, you start to build more infrastructure to make the software debuggable, robust, and flexible. In the resource-constrained environment, you'll need to determine how much infrastructure you can afford in terms of development time, RAM, code space, and processor cycles. What you started building initially is not what you will end up with when development is complete. And development is rarely ever complete.

Creating a system that is purpose-built for an application has an unfortunate side effect: the system might not support change as the application morphs. Engineering embedded systems is not just about strict constraints and the eventual life of the system. The 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 to accommodate whichever path the application takes. Get out your crystal ball.

Principles to Confront Those Challenges

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

Instead, imagine the puzzle has a time dimension that varies over its whole life: conception, prototyping, board bring-up, debugging, testing, release, maintenance, and repeat. Flexibility is not just about what the code can do right now, but also about how the code can handle its lifespan. Our goal is to be flexible enough to meet the product goals while dealing with the resource constraints and other 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 impact another area. This lets us take apart our system and put it back together a little differently when we need to.

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

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

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

Documenting your code is another way to reduce bugs. It can be 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.

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

Further Reading

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

- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley. There are many excellent references about design patterns, but this was the one that sparked the revolution. Due to its four collaborators, it is often known as the “Gang of Four” book (or a standard design pattern may be noted as a GoF pattern):
- Freeman, Eric T., Elisabeth Robson, Bert Bates, and Kathy Sierra. 2004. *Head First Design Patterns*. Cambridge, MA: O'Reilly Media.

Interview Question: Hello World

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

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

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



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

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

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

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

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

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 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.

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 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.

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—no matter whether 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 the processor is in the center, the communication objects are in the processor, and each peripheral is attached to a communication object.

For this example, I'm going to introduce a type of memory called flash memory. While the details aren't important here, it is a relatively inexpensive type of memory used in many devices. Many flash memory chips communicate over the SPI bus, a type of serial communication (discussed in more detail in [Chapter 6](#)). Most processors cannot execute code over SPI, so the flash is used for off-processor storage. Our schematic, shown at the top of [Figure 2-1](#), shows that we have some flash memory attached to our processor via SPI. Don't be intimidated by the schematic! We'll cover them in more detail in the next chapter.

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 (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.

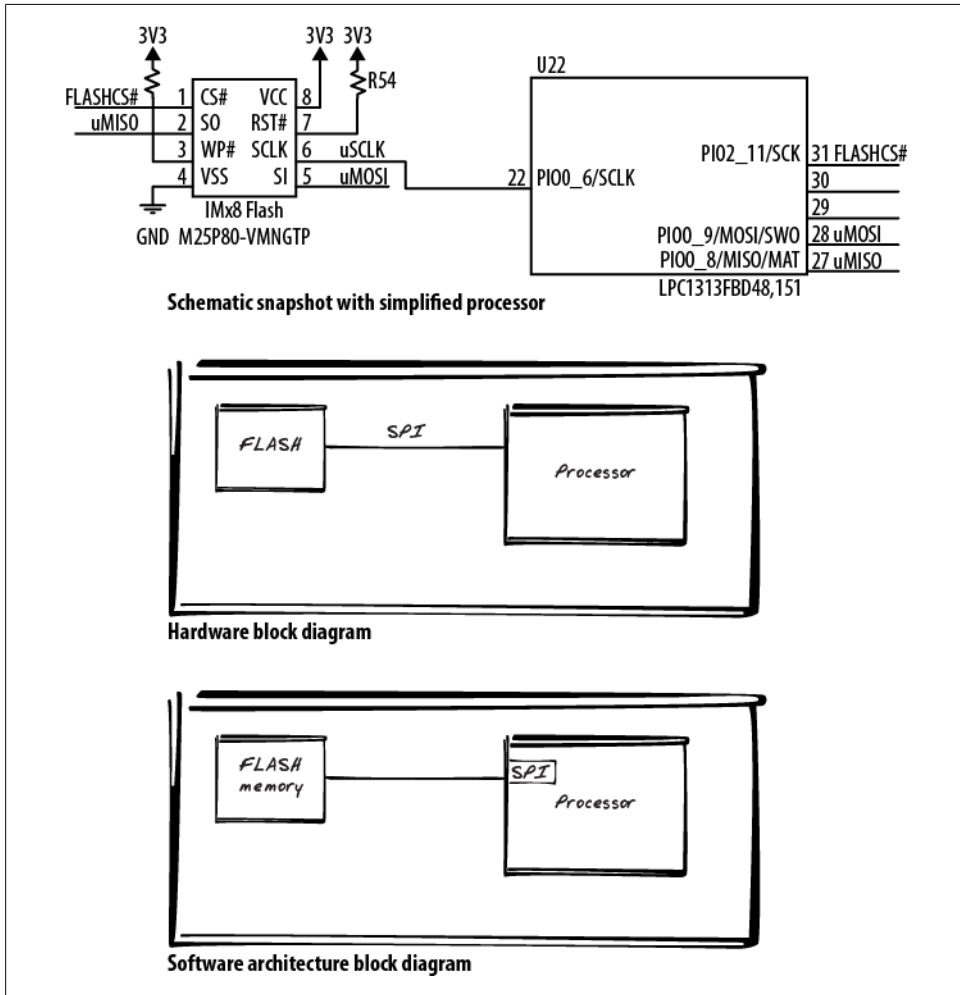


Figure 2-1. Comparison of schematic and initial software block diagram

The next stage is to add some higher-level functionality. What is each peripheral chip used for? This is simple 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 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 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 2-2](#).

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, another drawing often will 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'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.

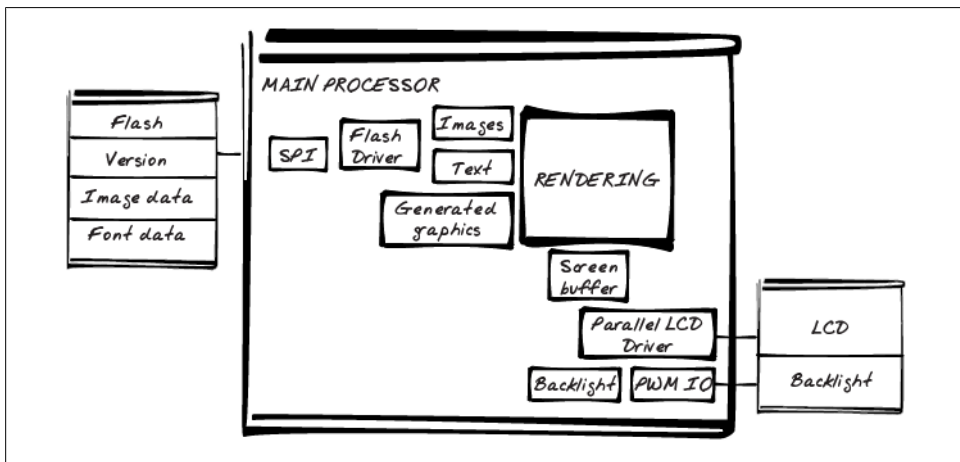


Figure 2-2. Software block diagram

Hierarchy of Control

The next type of software architecture diagram looks like an organizational chart, as shown in [Figure 2-3](#). It shows discrete components and which components call others. The whole system is a hierarchy, with “Main” at 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. 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).

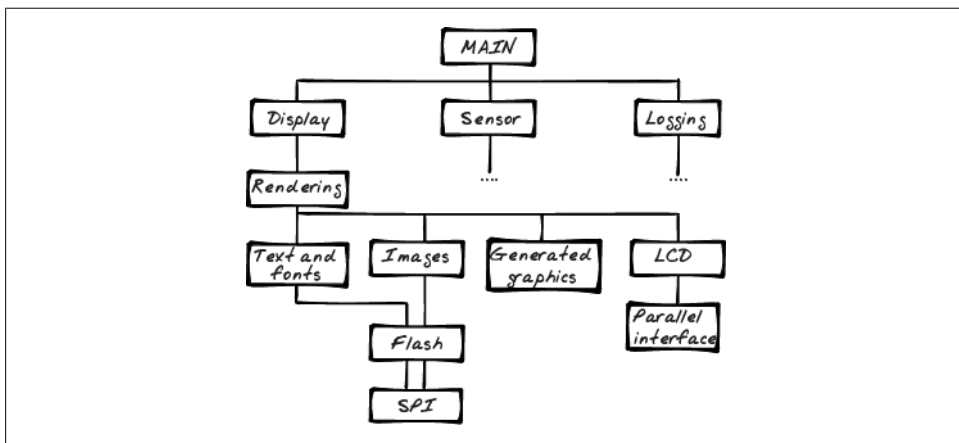


Figure 2-3. Organizational diagram of software architecture

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 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 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 have two threads or my display assets are used in an interrupt), the software will have to avoid collisions and any resulting corruption.

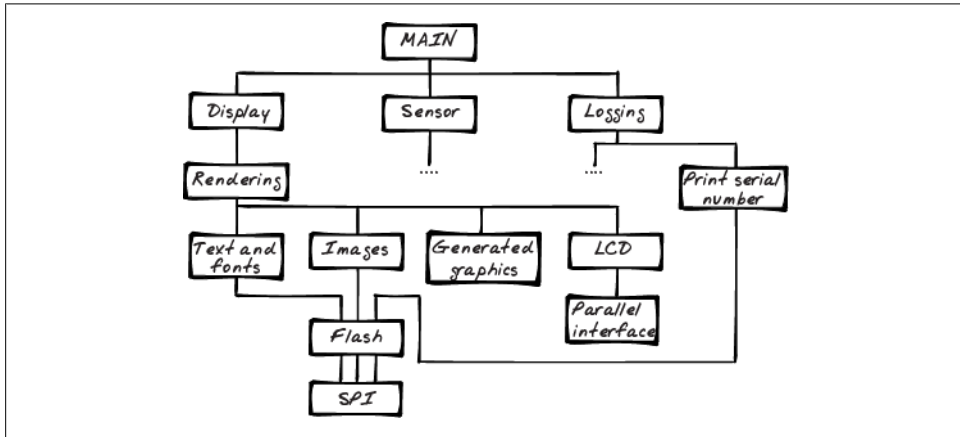


Figure 2-4. Organizational diagram with a shared resource

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 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.

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 (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 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 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 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. 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 maintainability.

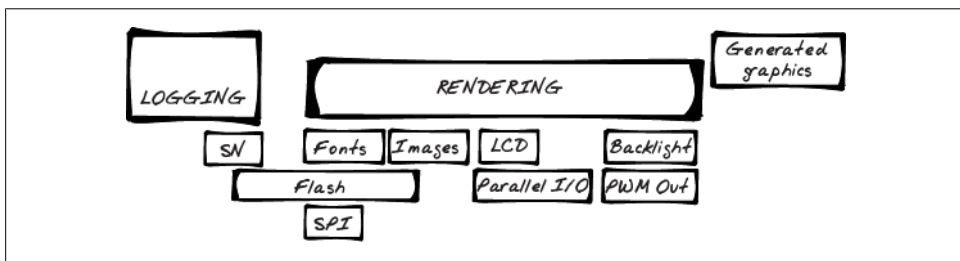


Figure 2-5. Layered software architecture diagram

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 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

Our 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 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 too much 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 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 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?
- 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 use your code or just for ease of testing).

In the example system, the LCD is attached to the parallel interface. In every diagram, it is simple, with no additional dependencies and no other subsystem requiring access

to the parallel interface. You don't need to expose the parallel interface; you can encapsulate it within the LCD module.

Conversely, imagine the system without the rendering module to encapsulate the display subsystem. The layered 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.

Because 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 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?