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 (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.

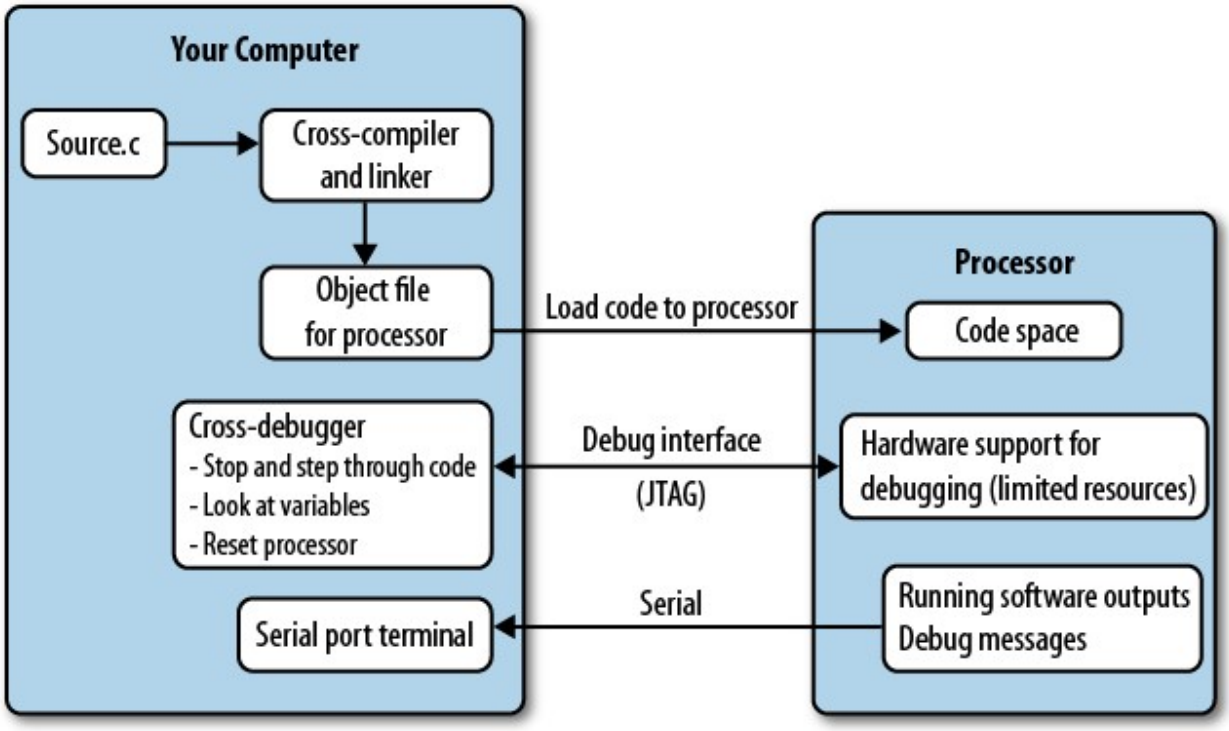

Figure 1-1. Computer and target processor

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 (somewhat 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).

TIP

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.